Operating System Verification — An Overview

Gerwin Klein

Sydney Research Lab., NICTA[⋆], Australia

School of Computer Science and Engineering, UNSW, Sydney, Australia

gerwin.klein@nicta.com.au

Abstract. This paper gives a high-level introduction to the topic of formal, interactive, machine-checked software verification in general, and the verification of operating systems code in particular. We survey the state of the art, the advantages and limitations of machine-checked code proofs, and describe two specific ongoing larger-scale verification projects in more detail.

Keywords: Formal Software Verification, Operating Systems, Theorem Proving

1 Introduction

The fastest, cheapest and easiest way to build something is properly the first time (Parker, 2007). This engineer's credo has made it into popular literature, but it seems to be largely ignored in the world of software development. In the late 1960s a software crisis was diagnosed on a summit in the Bavarian Alps (Naur and Randell, 1969): Software was getting more and more complex, and we had no structured way of building it. We ended up with projects that took significantly longer than planned, were more expensive than planned, delivered results that did not fully address customer needs, or at worst were useless. This summit in the late 1960s is widely seen as the birth of the discipline of *software engineering*.

Now, almost 40 years later, we have come a long way. There are numerous books on software engineering, a plenitude of methodologies, programming languages, and processes to choose from. We are routinely building large software applications, and often they work. The situation has certainly improved. Unfortunately, it has not improved enough. A large number of software projects still fail or produce software of low quality. The US National Institute of Standards and Technology estimated in 2002 that losses from poor software quality amounted to $59.5 billion (NIST, 2002) in the USA alone. The industries surveyed in greatest depth were aeroplane and car manufacturers as well as financial services. This value compares to a total of only $180 billion made in software sales in the USA in 2000.

The reasons for this situation are manifold and this article will not present a solution to fix the entire problem. Instead, this article aims to renew broad attention towards a method that achieves very high assurance of software implementation correctness for a comparatively cheap price: formal, mathematical proof. Formal methods held much promise in the 1970s but failed to deliver. It is our view that this situation has changed drastically. We now have the methods, tools, and people to achieve what was the goal then. Formal, machine-checked, mathematical proof that a

[⋆] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

given program correctly implements a formal specification is eminently achievable for important classes of software.

The main factors that influence the practicability of formal correctness proofs are the level of detail, the complexity, and the size of a program. The level of detail might be very abstract. For example, one might be interested in only whether the general idea of a high-level security policy works. In other projects, the level of detail might be much higher, including direct hardware interaction and low-level protocols, with the focus on showing that an implementation covers all possibilities and will work correctly under all circumstances. The second dimension, complexity, is the inherent complexity of the problem. Some problems can be encoded in restricted logics that offer a high degree of efficient automation, making verification as simple as the push of a button. Many interesting problems fall into this category. Other problems will require the use of more general logics where proofs need to be guided interactively by humans with the machine offering assistance and proof checking. The size of the program to be verified plays a crucial role. Depending on the nature of the problem the software solves, the effort needed for verification does not necessarily scale linearly, but sometimes exponentially or worse. Usually, size does matter.

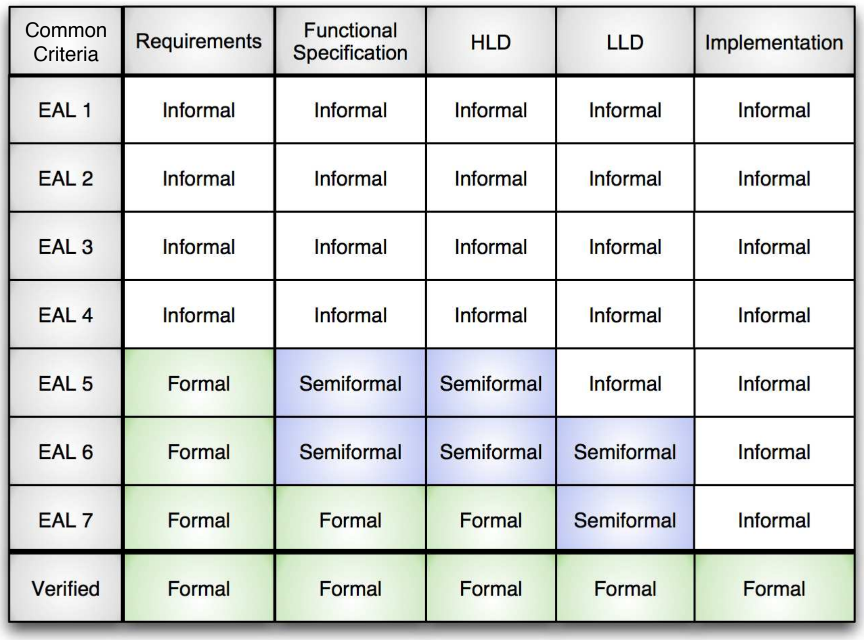
The exception is formal verification on very abstract levels. This usually considers only parts of software systems and would thus not be much constrained by the size dimension. Simple properties of high-level models are often easy to verify.

Formal verification of medium-level models of larger systems has been demonstrated before. These model the system fairly precisely, but not at the level of a C implementation. Achievable program size here would be in the area of 100,000 lines of code (loc) of the final system. The VerifiCard project (Jacobs et al., 2001), for instance, modeled the full JavaCard platform, including the source language, the compiler, and the JCVM machine language. Proofs included a number of complex security and safety properties of the platform and the correctness of the compiler. This was a major collaborative effort, but its results show that a very high degree of assurance at this level can be achieved. For comparison, full certification of the JavaCard platform to high security levels like Common Criteria EAL 7 seemed prohibitive to industry before this project, and was mostly abandoned. Now, Gemalto, for instance, commercially offers such a certified platform using essentially the same verification technology.

Formal verification of low-level implementations has so far been considered prohibitively expensive, or even impossible. In recent years, this view has been changing and there are a number of verification projects that target realistic amounts of low-level code. An obvious application area are Operating Systems (OS). The OS is at the core of almost every computer system, and a recent renewed trend in operating systems towards microkernels means that the size of the program to be verified is only around 10,000 loc. The properties to be considered range from fairly simple, e.g. memory safety, to very complex, e.g. implementing a full specification of behaviour.

It is this combination of low-level, complex property, roughly 10,000 loc that is still considered intractable in industry. Given the size of case studies that were published up to only 3 years ago, this impression is not surprising. However, it does not necessarily reflect the capabilities of the field. We aim to show that it is not only possible, but in fact can be cheaper to use formal verification instead of traditional methods for this area.

To get an impression of current industry best practice, we look at the Common Criteria (2006). The Common Criteria are a standard for software verification that is mutually recognised by a large number of countries. There are seven levels of assurance (EAL 1-7) in the standard and a number of so-called protection profiles that detail what exactly is being certified and for which application area. The standard is used for example by government agencies to specify software assurance requirements in requisitions and regulations. Fig. 1 shows a table with the assurance levels on the left, and software artefacts on the top.



Common Criteria	Requirements	Functional Specification	HLD	LLD	Implementation
EAL 1	Informal	Informal	Informal	Informal	Informal
EAL 2	Informal	Informal	Informal	Informal	Informal
EAL 3	Informal	Informal	Informal	Informal	Informal
EAL 4	Informal	Informal	Informal	Informal	Informal
EAL 5	Formal	Semiformal	Semiformal	Informal	Informal
EAL 6	Formal	Semiformal	Semiformal	Semiformal	Informal
EAL 7	Formal	Formal	Formal	Semiformal	Informal
Verified	Formal	Formal	Formal	Formal	Formal

Fig. 1. Common Criteria Evaluation Levels

From left to right, the software artefacts are: the software requirements, for example a security property, the functional specification, the high-level design of the system, the low-level design, and finally the implementation. The bottom row of the table compares this with software that is fully formally verified. The highest Common Criteria evaluation level requires a formal treatments of requirements, functional specification and high-level design. The low-level design may be treated semi-formally and correspondence between implementation and low-level design is usually affirmed in an informal way. To call a piece of software fully formally verified, the verification chain should reach at least down to the level of implementation. This article is about research projects trying to achieve this level of assurance.

The next section provides a more detailed overview of software verification in general, to define what *correct* means and to provide a bigger picture of what verification achieves.

Sect. 3 introduces microkernels, the subject of the verification efforts in this paper, and looks at which high-level properties and security policies can be enforced with these kernels.

Sect. 4 surveys the state of the art in the application area of OS verification. Next to early work in the area in the last century, there are a number of verification projects tackling the problem of verifying OS microkernels. Two of these are sufficiently resourced to achieve their goal.

2 Software Verification

This section gives a brief overview of software verification in general. For a more comprehensive, general-interest analysis of mechanised software verification, its development, and its role in building software, the author recommends MacKenzie (2001). For an overview on theorem proving

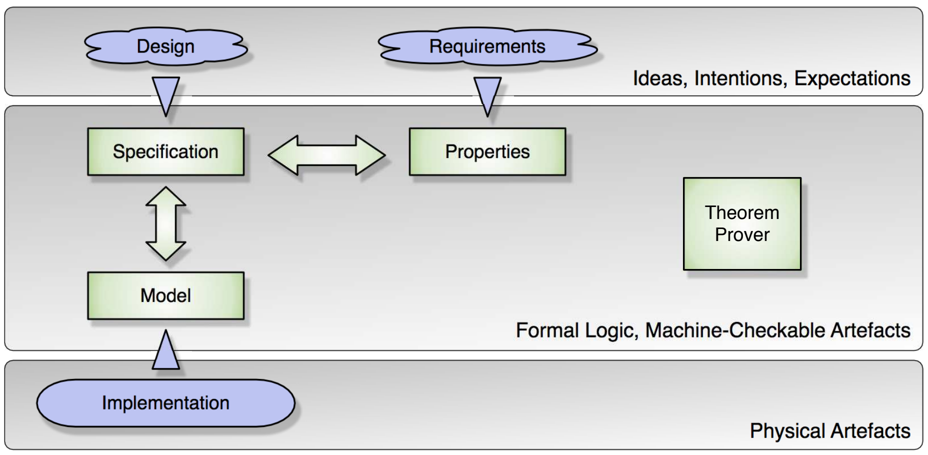


Fig. 2. Software Verification Artefacts

systems and what their proving styles look like, Wiedijk's compilation *The Seventeen Provers of the World* (Wiedijk, 2006) is an excellent resource. Wiedijk also maintains an online list of which of the 100 top mathematical theorems have been mechanised in a theorem prover (Wiedijk, 2008).

We begin the overview by investigating in more detail how formal verification fits into the bigger picture. We continue by looking at what formal verification promises, which will enable us to define more clearly what the promise entails, and we finish the overview by analysing the remaining risks.

2.1 The Big Picture

Fig. 2 depicts the main artefacts that are involved in software verification.

The top layer contains concepts and ideas. Examples of these are customer expectations, the mental image of how the system works in the minds of the developers, an informal requirements specification, or an informal, high-level design document. These are not constructs of logic, and they can not be reasoned about directly with mathematical logic, mechanised or otherwise.

These notional concepts have formal counterparts in the middle section of Fig. 2. Informal requirements, written down or existing just in the user's mind, may be translated into properties of the system specification. Examples of such requirements are *Applications never run in privileged mode of the hardware* or *User U may never access resource R unless the policy explicitly allows it*. Their formal counterparts would be expressions in a logic using terms and concepts that are defined in the formal system specification. For example, the system specification may define predicates for *users*, *resources*, *access*, and *policy*. The formal requirement then might be $\forall U \in users.\ \forall R \in resources.\ (U, R) \in access \longrightarrow (U, R) \in policy$. The system specification would further have some kind of formal description of how the system behaves, which actions it executes or how its state mutates over time.

The bottom layer of Fig. 2 shows the physical artefacts. Strictly speaking, the physical artefacts are the microchips, the transistors, their electrical charges, and the environment the machine operates in. Describing these precisely is the realm of physics, and if pursued to the end, the question of whether we can even describe and predict reality in full detail is a matter of philosophy. Fetzer (1988) uses this observation in a well-cited and much discussed, but ultimately pointless article to reject the whole idea of formal program verification as impossible. Following this view, we might just as well give up on any science that uses mathematics to model and predict reality. Pragmatically, the choice for this bottom layer still is a matter of lively discussion, together with the question of what a good formal model should therefore correspond to. The verification projects surveyed in the second part of this article start from different levels of abstraction with different aims and choices. Some examples for what is considered the physical artefact are C source code, assembly source code, assembly binary code including memory contents, or a VHDL description of the machine. Program source code could be viewed as a physical artefact, and it is usually the lowest level a software development project deals with. Nevertheless, if one aims to eliminate all possible sources of error in an implementation, there is a large gap between the source code and the physical memory state of a computer—it involves at least the compiler, the linker, the operating system, and input devices. Even a gate level description of the microprocessor might still not be precise enough to capture reality fully. Where one makes the cut-off depends on what level of assurance one is interested in and where a point of diminishing returns is reached.

Although often confused with formal artefacts, all of the above-mentioned concepts (program source code, assembly code, gate level description) are not formal yet. They may be very close to formal artefacts, and in an ideal world they even might be, but currently they are merely inputs for tools or machines that we use in development. We still cannot reason about them directly. For that to be possible, we again need a translation into formal logic, preferably in the same formal framework that we use for specification and requirements. For example, in this context, even a VHDL description itself would not be a formal artefact, whereas its translation into the formalism of the theorem prover would be. The distinction becomes clearer for higher-level programming languages. An example of this would be a formal semantics of the C programming language that assigns logical meaning to each of the syntactic constructs of the language, for instance describing how each statement changes the machine state. Although Norrish (1998) comes close, there currently exists no such semantics for the full C language. Currently, programs as such are clearly not formal artefacts.

It is only at this point, with the translations from ideal concepts and from the physical world into formal logic, that we can use the tool of formal proof. In an ideal setting, at this point we are able to make formal correctness guarantees with an absolute degree of assurance. Even in the less than perfect real world, with machine-checked proof we can achieve a level of assurance about our formal statements that surpasses our confidence in mathematical theorems which we base all of physics and engineering on. Not trusting these makes as much sense as not trusting the theorem of Pythagoras.

There exist several different kinds of formal methods that may be used to prove the connections between properties, specification, and model. They range from fully automatic methods that may even construct the system model automatically from source code to interactive methods that require human guidance. The trade-off is between the expressiveness of the logic and the degree of automation. Logics with little expressiveness, e.g. propositional logic, make it hard to write down the three required logical artefacts and hard to understand the resulting formalisations, but they offer a high degree of automation in return. So-called SAT (satisfiability) solvers are such successful, automated tools. Logics with a high degree of expressiveness like Zermelo-Fraenkel Set Theory or Higher-Order Logic (HOL) make it easier to express properties and specifications precisely and more importantly in a readable way, but they require human creativity and expertise in performing the proof. There are many steps in between these two extremes with model checking and automated first-order theorem proving as two intermediate examples. As we shall see in later sections, all of these have their use in the analysis of operating systems. It is currently mainly the interactive methods that provide sufficient expressivity to handle the formal statement of full functional correctness of an OS microkernel, although there are efforts to shift these proofs towards more automation where possible.

It is illustrative to view the traditional methods of code inspection and testing in terms of Fig. 2. Code inspection means looking at the code, following its logic, and manually comparing it to the intended behaviour. Testing means implementing the program, running it on input data, and comparing the results to the expected outcome. Code inspection attempts to make a direct connection between the ideal model of requirements and design that exists in the head of the inspector and the arguably physical artefact of source code. To accomplish this, the code inspector does at least need to construct a mental model of the physical system behaviour which in spirit, but not in form, is usually very close to what a formal system model might be. One might say that code inspection requires a mental model of system behaviour, but not an additional model of the requirements. Testing similarly bypasses formal models, but again not always entirely. Depending on the particular testing methodology, models of system behaviour, expected output, code coverage and more may be used. These correspond to the high-level system specification (coverage) and requirements (test sets). Here, one might say that testing does not require a physical execution model because it runs the physical system, but it does require some kind of model of correct system behaviour.

2.2 The Promise

The basic promise of formal verification is the following.

Make sure software works.
Convince others that it does.

The first part of this promise concerns software correctness. As we will see in the next section, we need to be more precise at this point. Correct software can still be useless. The common meaning of *works* is that the software fulfils the user's or customer's requirements and expectations. The more specialised meaning often employed in the context of verifying software (formally or otherwise), is that the implementation fulfils its specification.

There are a number of techniques that can be used to achieve this kind of correctness. Current industry best practice comes down to code inspection and testing. The alternative is formal verification. Formal verification is used in industry, but not nearly as widely as testing and inspection. We will analyse specific advantages and disadvantages of these three approaches further below.

The second part of the promise is possibly even more interesting than the first. It is not enough to build a piece of software and be confident that it works as intended. One needs to be able to demonstrate that it does work and convince others of the fact. If you are building a new model of airplane that will transport millions of people during its product cycle lifetime, you need to be able to convince the relevant authority such as the FAA that the plane and its software are safe. If you are building a storage device for secret information, you need to convince software certification bodies like the NSA that the secret cannot be leaked.

The process of convincing others of software correctness is most prominent in software certification. Currently, these software certifications are process based, that is, they are mostly concerned with how the software was developed, and for sensitive projects even with who worked on it, which methods were used, and which tests and inspections were conducted. Certification authorities mandate a large amount of documentation that does not deal with the final software artefact itself, but rather with the software development process instead. One reason for this is that the quality of tests and inspections depends very much on organisational discipline and the person who conducts them. This is obvious for code inspections, but applies to tests as well: Given an extremely large input data space to choose from, it takes great skill to design and implement useful, repeatable tests that build confidence in the correctness of the software. An old adage of formal methods is that for any interesting system, tests can only show the presence of bugs, not their absence. Proofs, on the other hand, can.

Some certification processes, like Common Criteria, do require formal verification, but only on abstract levels, and only with an informal, usually inspection-based connection to the implementation.

The main difference between formal proofs and current certification processes is that proofs are artefact-based. With a machine-checkable proof of correctness, it does not matter which process was used to implement the software. Neither does it matter who worked on it, or even who worked on the proof and how the proof was arrived at. To check the validity of a proof, only the formal specifications and the proof are needed. What remains is validating that the formal models indeed describe the physical artefact and requirements correctly. This is independent of the development process itself and shifts effort into the early requirements part of the development process, where studies (Bowen and Hinchey, 2005) say the money should be spent. Shifting from process-based to artefact-based certification has an immense potential for simplification, saving costs, and making certification more meaningful at the same time.

2.3 What could possibly go wrong?

The previous two sections showed software verification from a high level and explained what it is trying to achieve. This section analyses the remaining risks. After the promise of an absolute, mathematical guarantee of correctness, the question insinuates itself *Surely there this a catch? Life cannot possibly be that simple.*

There are four main points of possible failure: translating each of the three informal concepts into their formal counterparts and the correctness of the proof itself.

The latter point of proof correctness is within the realm of formal logic and has been studied extensively in the field. Proofs about large systems are large, complex, highly detailed, and in many parts repetitive. This is one of the worst possible combinations for human error (Reason, 1990). It is therefore reasonable to expect errors in large proofs, especially if they are constructed by hand. The solution to this problem is to mechanise proofs, that is, to let a machine check or even construct the proof. In both cases, the immediate question is *What if the theorem prover is incorrect?* There are a number of answers to this question depending on how seriously this concern is taken and whether the human user of the theorem prover is seen as benign or potentially malicious.

Careful implementation. A large number of formal methods tools derive their trust from having implemented their tool carefully and having based it on systematically, often formally analysed algorithms. Tools in this category are PVS (Owre et al., 1996), ACL2 (Kaufmann et al., 2000), the B-tool (Abrial, 1996), and most popular model checkers, first-order automatic provers and static analysis tools. They assume a benign user who would not actively exploit loopholes in the implementation and upon noticing an implementation error, would either fix or avoid the problem. This is based on the fact that prover implementation errors are second-order errors, and have only a second-order effect: Most such bugs do not lead to proofs that are wrongly claimed to be correct, but rather lead to the user not being able to prove a true property. Even in the case of wrongly proving an unsound statement, this is usually noticed very quickly, because complex properties are suddenly proved easily. Although the author of this article is not aware of any catastrophic software failure that can be traced back to a proving tool itself being incorrect, this is not necessarily a satisfactory answer for high-assurance software certification unless the tool itself has undergone an in-depth traditional certification process.

LCF-style provers. There is a family of theorem provers that uses a concept from LCF, Scott's logic of computable functions (Scott, 1970), in the mechanisation of proofs: correctness by composition. The prover at its core has only a very small kernel of functions that are able to create new theorems. These functions correspond directly to the basic axioms and derivation rules of the logic that is being implemented. The theorem prover then relies on mechanisms of the programming language to ensure that all other parts of the system have to invoke kernel functions to create more complex building blocks such as automated proof search. This means that only implementation errors in the kernel can be critical to soundness. Bugs in any other part of the system can only lead to frustration on the user's part when a true theorem cannot be proved, but never to an unsound statement being treated as a theorem. The idea is that this proof kernel is small enough to inspect and certify satisfactorily by traditional means. Tools in this category are HOL4 (Norrish and Slind, 1998–2006), Isabelle (Nipkow et al., 2002), and HOL-light (Harrison, 1996).

Independent proof checkers. Another approach to the correctness of theorem provers is the observation that proof checking is much easier than proof search and construction. Consequently, upon proving a theorem, an external standardised proof representation (usually a proof term) can be created and later checked independently by a small and easy-to-certify proof checker. Incidentally, LCF-style provers can be easily extended this way, because their proof kernel already provides most of the necessary infrastructure. Isabelle, HOL4, and HOL-light have been extended with proof term generation. Other tools in this category are often found in the area of constructive logic, where the prover additionally exploits the fact that the constructive proof of a formula can be turned directly into an algorithm that computes the effective content of the formula being proved. Two tools in this category are Coq (Bertot and Castran, 2004) and Minlog (Schwichtenberg, 2004).

Verified theorem provers. One suggestion that is often immediately put forward whenever the correctness of theorem provers is pondered is *Why not use the theorem prover to formally verify itself?* As John Harrison put it when presenting a paper on just such an effort: This endeavour would be impossible and useless at the same time (Harrison, 2006). Nevertheless it is fun. It is impossible, because ultimately one hits Gödel's incompleteness theorem (Gödel, 1931) which states that any logic expressive enough to prove its own consistency is necessarily inconsistent. That means we should hope that it is impossible to formalise the logical content of the theorem prover in its own logic. If it is not impossible, it is already useless, because the logic the theorem prover implements has to be unsound. Harrison deals with this problem by slightly changing the logic used to verify the theorem prover, but using the same prover infrastructure. Changing the axiom system to a similar, but slightly different one allows him to escape the Gödelian restriction. Having so evaded the *impossible* part, strictly speaking, one still gets caught out with a useless statement. There is no guarantee that the new axiom system is sound and even if it was, we still have used the tool itself to show its own correctness, which means we are still relying on its infrastructure to be correct. A soundness-critical bug in the infrastructure can easily show its own correctness by exploiting precisely this bug. Nevertheless, going back to the argument of benign users, it is an interesting and revealing exercise to go through the process that genuinely increases the degree of assurance of the theorem prover being correct. Additionally checking the proof in a different, compatible prover like Isabelle/HOL makes the probability of accidentally using an infrastructure implementation bug very small. There is currently only one such self-verified tool: Harrison's HOL-light. A similar idea is using one theorem prover to show the correctness of another. Followed to the end, this must lead to either an infinite chain of such proofs or stop somewhere at a traditionally certified system. The author is not aware of any production-quality systems that have undergone such a process, but there are certainly examples of theorem-proving algorithms and models being analysed in theorem provers (Ridge, 2004).

With independent, small proof checkers, even the most paranoid customers of formal verification, such as the defence and intelligence communities, are satisfied that the correctness of formal proofs themselves is essentially a solved problem. Much more critical and potentially dangerous are the relationships between the informal and formal parts. In the following, we look at the formal/physical and formal/idealistic dimensions in turn.

Translating physical artefacts into formal models. One crucial question in formal verification is how closely the model resembles the physical system. The exposition in Sect. 2.1 has already established that this connection can never be fully precise. As in all of engineering, we work with abstractions of the real world. The important part is to cover what is relevant to the task at hand. As mentioned before, opinions vary on what is appropriate here, because the more detailed the model, the harder and more resource-intensive the verification. Even if it is agreed what the appropriate level should be, it is still possible to make mistakes in the translation itself. If for instance, the semantics of the C programming language on the formal side does not match the compiler's interpretation, then the proof constructed with that semantics will exhibit a large gap to the physical artefact and might make wrong predictions of its behaviour. This gap is often downplayed in academic literature and in the marketing material of companies. For instance, there currently exists no operating system kernel that can be called formally verified in the above sense, although two of the projects surveyed below come close and have made promising progress towards that goal. Not even the highest Common Criteria evaluation level (EAL 7) demands a formal or even semi-formal implementation. No strong, formal case needs to be made that the low-level design adequately models the implementation. What is verified is the low-level design with respect to the functional specification only. This, of course, is already a very worthwhile process, and it might be all that is required or appropriate for a specific application, but it falls short of what can be achieved. Even though the projects surveyed below have different views on which level of implementation detail is appropriate, they all agree that it should at least be as deep as the source code level of the implementation language. This level is significant, because it is the lowest level that is usually worked on manually. Deeper levels are commonly produced by automated tools such as compilers and while they may produce errors, they at least do so systematically.[1]

Translating requirements and design into formal specifications and properties. The gap between the ideal and the formal world is often hard to even define. One aspect of this gap is an almost banal question of notation: does the written formula actually mean what the reader thinks it does and does this adequately captures the requirement or design idea of the developer? Since proofs are machine-checked, this is where most of the remaining risk is concentrated. For large systems, it is deceptively simple to make inconsistent statements that are trivially true. For instance, the difference between $\forall x. \exists y. P(x, y)$ and $\exists x. \forall y. P(x, y)$ might seem small and subtle to the untrained eye, but can of course make a world of difference in a specification. As in mathematics, adequate notation, using a logic of appropriate expressiveness and short, clear property statements are essential. In addition to property statements, it is also important to get the high-level description of the system right. If at all possible, it should be in a form that can be read, explored or even executed by the person designing the system. Otherwise one might build a perfectly correct coffee

[1] Edsger Dijkstra is attributed with saying *We must not put mistakes into programs because of sloppiness, we have to do it systematically and with care*.

machine that satisfies all security requirements when the customer actually wanted a cruise missile guidance system instead.

The main lesson to draw from this section is that formal verification is able to tell that a program was built right, but it cannot tell that the right program was built. The latter is the job of validation, and it is important to understand that verification without validation is not a strong proposition. Validation is where testing and inspection are of real value. If the formal model is executable, testing can be used with a high degree of automation to validate the translation between the physical and the formal world. Prototyping and testing on executable high-level models can also be used to bridge the gap between the ideal world of expectations and a formal specification, quickly exposing inconsistent property statements. Inspection is best used at the level of high-level properties and specifications to make sure that the formulas correspond to intentions and requirements. Verification can handle everything in between.

Thus, very much like Bowen and Hinchey (2005), we do not advocate verification as the sole solution to the problem of making complex software work, but we advocate that testing, inspection, and verification each be used in a complementary way, each at where they bring the largest benefit.

3 Microkernels

This section introduces the subject of the verification projects in this article: operating system microkernels. By definition, the kernel of an operating system is the part of the software that runs in the privileged mode of the hardware. This privileged mode typically distinguishes itself from normal user mode by additional machine instructions and by write access to hardware registers and devices that are invisible or read-only in user mode, for instance the memory management unit (MMU) of the processor.

Microkernels try to minimise the amount of code that runs in the privileged mode. There are many kinds of such small OS kernels. For the purposes of this survey, we restrict ourselves to kernels such as Mach (Rashid et al., 1989), L4 (Liedtke, 1995), and VAMOS (Gargano et al., 2005) that do not rely on co-operation from user mode processes to implement their services. This usually implies that the kernel at least manages the MMU and runs only on hardware that provides such a unit. The MMU can be used to implement virtual memory. It provides translation from virtual to physical addresses and other management functions like memory access rights (read/write/execute). If operated correctly, it can be used to protect kernel memory and memory of different user applications from each other.

Sect. 3.1 gives sample architectures of microkernel-based systems. In Sect. 3.2 we investigate the properties and policies these kernels provide, which will become the verification targets of Sect. 4.

3.1 Architectures

Fig. 3 compares traditional monolithic kernels such as Windows and Linux on the left with the microkernel approach on the right. In a monolithic kernel, device drivers, system services like the file system, and even parts of the graphical user interface are part of the kernel and run in privileged mode. In the microkernel approach, only the microkernel itself runs in privileged mode. It exports sufficiently powerful mechanisms to run system services and even low-level device drivers with the same degree of protection that is afforded to applications.

This has several advantages for system design and reliability. In a monolithic kernel, even an unimportant part of the kernel, e.g. a graphics card driver on a network server, has full access to all

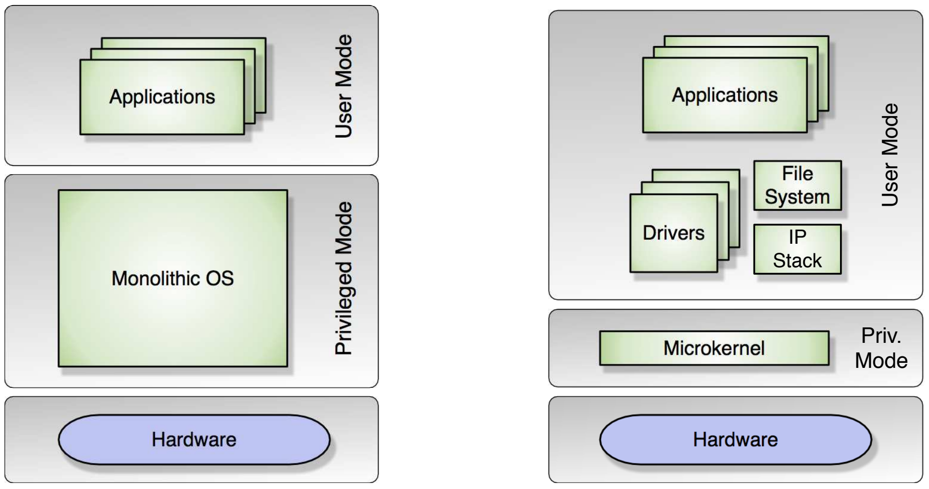


Fig. 3. Example system architectures.

memory and all kernel data structures. A fault in such a driver can be exploited to fully undermine the whole system, be it maliciously or by accident. This is not a theoretical threat. Such exploits are found frequently in the wild, see e.g. Slashdot (2006) or Securityfocus (2008). In a microkernel on the other hand, such a driver would run in its own protection domain. An exploit in the driver would compromise only memory of that driver and will only deny service to those parts of the system that rely on it. For the normal operation of a network server for instance, the graphics driver is irrelevant and could be safely terminated and restarted. The main disadvantage of the microkernel approach was famously brought forward by Linus Torvalds in 1992 (DiBona et al., 1999): Performance. Creating these protection domains, switching between them for execution, and passing messages across domain boundaries was considered expensive overhead. This overhead would be incurred at the innermost, lowest level and therefore accumulate rapidly and unacceptably for all OS services. Providing roughly equal services, the main selling point of operating systems is performance, and the years that followed seemed to prove Torvalds right. Monolithic OS kernels not only persevered, but rose in popularity.

More recently, however, it has been shown that the performance of microkernels can be improved to minimise this overhead down to the order of magnitude of a C procedure call. Especially the L4 microkernel is known for its exceptional performance (Liedtke, 1995). High-performance user-level network device drivers have been created Leslie et al. (2005) and even fully a virtualised Linux runs on top of L4 with only a very small performance impact (Härtig et al., 1998). Consequently, industry interest in this system architecture is on the rise again. The most common use scenario for microkernels is not the fully decomposed OS of Fig. 3, though. Instead, microkernels have achieved success as powerful para-virtualisation engines, running for example multiple copies of a guest OS like Linux on the same machine with protection boundaries between the instances. In para-virtualisation, small parts of the guest OS are modified to call microkernel primitives instead of their own mechanisms. In light of Torvalds' performance complaints, this lead to amusing effects such as a para-virtualised L4/Linux which runs certain context-switching benchmarks thirty times faster than native Linux (NICTA, 2006). This could be achieved because Linux's context-switching code was replaced with a call to the highly performance-optimised version of L4. Another example in widespread use is Apple Computer's Darwin Mac OS X which runs as a largely monolithic OS with a modified Mach kernel. The resulting design is too large

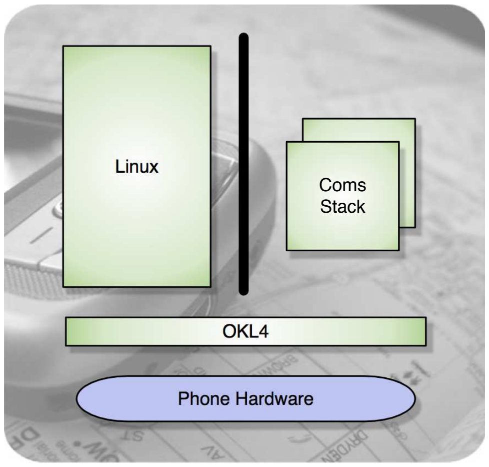


Fig. 4. Example system architecture with OKL4.

to still be called a microkernel, though. Close to the original vision of microkernels is the OKL4 kernel, a commercial development of the L4 kernel currently running on 100 million 3G mobile phones, an estimated 30% of the global market. Fig. 4 shows a common system architecture for this kernel.

The left half of the system runs the phone user interface on top of a traditional phone OS like Windows CE or Embedded Linux. Isolated from this user interface runs the phone communications protocol stack. Only explicitly enabled communication channels between the phone stack and the interface are permitted by the kernel. This allows chipset manufacturers to run their communication protocols in software without having to open their intellectual property to the OS manufacturer. It also protects the phone infrastructure from software exploits and the notorious instability of current phone user interfaces. This protection is critical, because a suitably exploited communications stack on the phone is sufficient to take down the operation of a whole cell of a network provider who may be liable to provide guaranteed uptime for emergency services.

As becomes clear from the discussion above, a wide variety of systems and architectures can be implemented with microkernels. One topic of discussion in the OS community is whether a microkernel should provide a hardware abstraction layer (HAL) and hide hardware details, or simply provide a set of mechanisms to control the hardware and export underlying complexities to user space. The verification projects surveyed in the next section take different views on this topic which influences what the formal models look like and what properties are being proved.

HAL or not, all microkernels used in these verification projects provide at least the following services:

Threads. Threads are an abstraction from the CPU, in effect allowing multiple programs to share a single CPU and provide the illusion of parallel execution. Typical operations on threads are starting and stopping execution, managing scheduling parameters or inspecting and possibly modifying the execution state of a thread.

IPC. IPC stands for inter process communication. This communication can be synchronous (communication partners have to rendezvous and be ready at the same time) or asynchronous (which usually involves some form of buffering or potential message loss). IPC is the central mechanism for transferring information across protection boundaries.

Virtual Memory. As mentioned above, protection domains are usually implemented using MMU hardware mechanisms. The kernel can either fully manage the MMU or just export sufficient mechanisms and resource arbitration to user space for implementing specific virtual memory policies there. In the latter case, the mechanism needs to be hardened against malicious use from user space.

The typical implementation size of a microkernel is on the order of 10,000 lines of C code. The kernels differ in which additional services they provide and how access to shared resources such as interrupts and devices is managed. We will go into more detail where necessary in later sections when we look at specific verification projects.

3.2 Services and Policies

The usage scenarios and consequently the services and mechanisms the kernels are optimised to provide are as diverse as the microkernels themselves. Without attempting to create a full taxonomy of microkernels, we can divide them into kernels intended to support general-purpose operating systems and kernels that are optimised for smaller, embedded systems. The emphasis in the former case is often on abstraction and flexibility, such as the ability to dynamically reconfigure possibly hierarchical security policies and protection domains. The emphasis for embedded systems on the other hand is often on small memory footprint, short interrupt latencies, real-time capabilities and strong isolation guarantees. There are also systems that cater to both areas.

The most complex end of the scale tend to be general-purpose kernels. The least complex end of the scale usually are specific-purpose embedded systems kernels. An example would be an extremely simple separation kernel whose only purpose is to multiplex hardware into a fixed number of partitions without any communication between them. These kernels would not even provide the IPC service mentioned above. Allowing controlled communication would be the next step up; allowing communication channels to be changed during runtime would further increase complexity. Allowing to adjust protection domains at runtime, or to create new protection domains at runtime increases complexity again, until we arrive at a fully dynamic system where communication, CPU, or memory access cannot only be changed dynamically, but the authority to do so can also be safely delegated to subsystems.

Functional specifications of microkernels usually consider at least two levels. The first level concerns all the services the kernel provides, and their formalisation. In order to be useful and general enough, this kind of specification needs to be fairly detailed and describe precisely which actions the kernel can perform and what their effect is. The second, more abstract level concerns a specific purpose important for the kernel. For microkernels, these tend to be security and isolation properties. If the system can be reduced to its security aspects only, the specifications involved become much smaller, more abstract and easier to reason about.

In the following, we concentrate on security-related properties that make interesting targets for formal verification.

Access Control and Data Separation A classic target for verification is access control. One purpose of the kernel will be to arbitrate and restrict access to shared resources such as memory, devices and CPUs. There is a whole host of different abstract access control models that kernels can implement and that could be used as a top-level specification. Examples are capability models (Dennis and Van Horn, 1966) or the Bell-LaPadula model (Bell and LaPadula, 1973). In the first, access to a resource is granted only on presentation of a sufficiently authorised capability to that resource. A capability is basically a set of access rights together with a reference to a specific resource. Capability systems allow very fine-grained access control.

Security-critical systems often require *mandatory* access control. This means that the kernel enforces a security policy via its access control method and does not rely on good behaviour of processes and users not to circumvent the policy. Mandatory access control is used to implement system-wide security policies, for example stating that files with classification level top-secret have to remain protected. In contrast to that, *discretionary* access control allows users to make policy decisions themselves and for instance grant read access to everyone; an example is the standard Unix file system model.

Access control can be used to achieve data separation between processes. Data separation means that no process is allowed access to memory of other processes, be it by accident or malicious intent. Because they are easy to formalise and already provide strong security implications, access control models are the most popular specification target for the verification projects surveyed below.

Information Flow A stronger property is control of information flows. Formally, it is hard to define what information flow is, and there currently is still no universally accepted definition. Information flow properties are usually about secrecy: Process A stores a secret cryptographic key, and no other process should be able to infer what that key is. The information is not allowed to flow outside process A. Pure access control models are not strong enough to establish this property. They will prohibit direct access to the memory storing the key, and might also prohibit explicit communication channels, but there might be indirect means of inferring what the key is. One of the most challenging scenarios is collaboration between security levels to break system policies. For example, top-secret process A might be controlled by a spy in the organisation, and unclassified process B might be the means to send the information outside. In this case, so-called covert channels need to be considered: process A could for instance use up its full time slice for transmitting a 1-bit and otherwise donate its remaining time slice. All B needs to observe is if more than its allocated time is available for computation. If yes, then A has transferred information. Of course, one can devise mechanisms against this specific attack, but the game becomes one of guessing the attack and guarding against it. Creating a new attack 'merely' requires looking at the model and choosing something that is below its level of abstraction or that is otherwise not covered. The above example is a timing channel attack; it would not work against a kernel providing strict temporal separation, such that no thread can influence the scheduling and computing time available to other threads.

Other attacks include observing CPU temperature, memory bus congestion, or cache misses. Known attacks against smart cards go as far as using radiation or heat to introduce memory faults into a computation, using an electron microscope to look at the chip in operation, etc. Modern high-security smart cards have physical tamper-proof devices against such attacks. It is unclear how far software alone can guard against them.

Non-Interference More well-defined than general information flow is the property of non-interference (Goguen and Meseguer, 1982, 1984; Rushby, 1992). This property is able to capture indirect information flows in a model if the code for both sides is known. Given a secret process S and an attacker A, it requires one to prove that the behaviour of A cannot depend on the actions of S, i.e. that S does not interfere with the behaviour of A. It is possible to establish this property by looking at execution traces of the system: Given a trace where execution of both S and A occurs, one needs to show that the result in A is the same when all S actions are removed from the trace. Unfortunately, this property is not necessarily preserved by the proof technique of refinement which is the one most commonly used to relate an implementation to its model (Jacob, 1989). This means that proving non-interference of the model only implies non-interference of the implementation when special care is taken in the implementation proof. The standard access control models do not have this problem.

Common Criteria The Common Criteria introduced in Sect. 1 have a protection profile geared towards separation kernels which are close to the services microkernels provide. The properties considered important for this class of kernels are: Non-bypassable, Evaluatable, Always invoked, Tamper proof (NEAT). They are designed to be able to build systems with multiple independent levels of security (MILS), i.e. systems with components of different trust and assurance levels. In this context, *non-bypassable* means that the communication channels provided by the kernel are the only ones available to components and that there are no lower-level mechanisms that could be used for communication. *Evaluatable* means that the system must be designed such that an in-depth evaluation, including a fully formal evaluation if required, is possible. This restricts the size, complexity, and design of such systems. The term *always invoked* means that the access control methods of the kernel must be used every time a resource is accessed, not for instance just the first time a resource is requested. Finally, *tamper proof* means that the access control system cannot be subverted, e.g. access rights be changed against the security policy by exploiting a loophole in the mechanism. These concepts, although intuitive, are not necessarily easy to formalise and prove directly. However, they can be established with reasonable effort by inspecting high-level formal models of the system. In this case it is important that the proof method used to relate the high-level model to code preserves these properties down to the code level.

Because of their small size, the small number of services they provide, and their critical nature, microkernels offer a high-yield target for pervasive formal verification. The next section surveys the state of the art and the progress of current microkernel verification projects.

4 OS Verification – The State of the Art

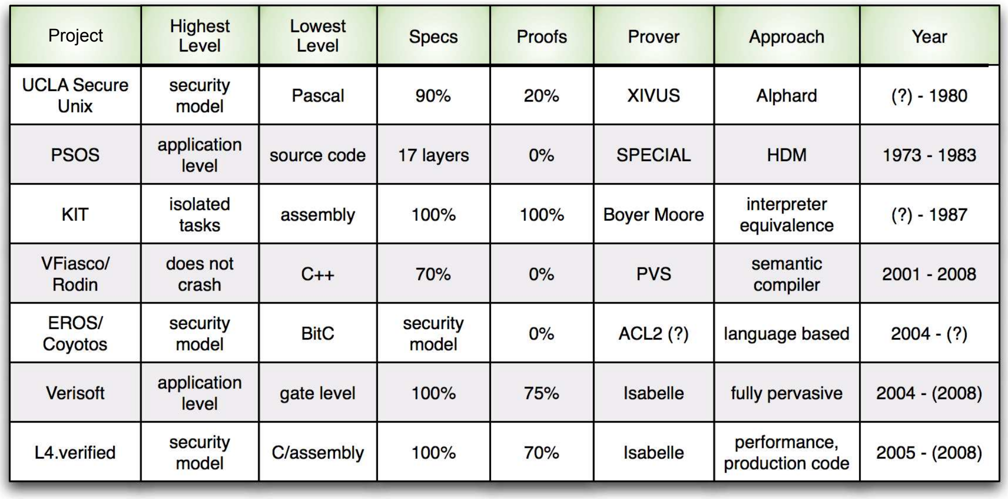
We begin the survey with a high-level, table-style overview. The following sections then go into more detail about early work on OS verification and the current projects.

We limit the survey to projects that at least aim to prove high-level functional correctness or security properties of an OS or OS kernel implementation. The kernel should be usable for a general-purpose system. To keep the scope manageable, we do not include efforts such as SLAM (Ball et al., 2006) or Terminator (Cook et al., 2006, 2007) which focus only on specific aspects of correctness such as deadlock freedom and termination. These are certainly worthwhile efforts, but by themselves they would not produce a formally verified operating system kernel.

4.1 Overview

The table in Fig. 5 gives a condensed overview of the projects surveyed. The first two columns after the name of the project contain the highest and lowest levels of verification artefacts the project considers. They usually correspond to the terms Specification and Model of the verification picture in Fig. 2. The next two columns detail what percentage of the specification artefacts in the project have been completed, and how much of the proofs are estimated to be discharged. The sixth column gives the main theorem proving tool used. The penultimate column contains one or two key phrases indicating the approach used, and the last column indicates when the project took place. Years in parentheses are estimated completion dates.

The first three projects in Fig. 5, UCLA Secure Unix, PSOS, and KIT, concern early work in the 1970s and 1980s. While they pioneered the field and all finished successfully in their way,



Project	Highest Level	Lowest Level	Specs	Proofs	Prover	Approach	Year
UCLA Secure Unix	security model	Pascal	90%	20%	XIVUS	Alphard	(?) - 1980
PSOS	application level	source code	17 layers	0%	SPECIAL	HDM	1973 - 1983
KIT	isolated tasks	assembly	100%	100%	Boyer Moore	interpreter equivalence	(?) - 1987
VFiasco/ Rodin	does not crash	C++	70%	0%	PVS	semantic compiler	2001 - 2008
EROS/ Coyotos	security model	BitC	security model	0%	ACL2 (?)	language based	2004 - (?)
Verisoft	application level	gate level	100%	75%	Isabelle	fully pervasive	2004 - (2008)
L4.verified	security model	C/assembly	100%	70%	Isabelle	performance, production code	2005 - (2008)

Fig. 5. OS Verification Projects.

none of them produced a realistic OS kernel with full implementation proofs. We look at these projects in more detail in Sect. 4.3.

The next two projects in the table are recent, smaller-scale efforts that have made significant contributions. The first one, VFiasco, is a project at TU Dresden, Germany, and RU Nijmegen in the Netherlands. It is described in Sect. 4.4. The second project, the verification of the Coyotos kernel, has unfortunately been cut short. We survey its contributions in Sect. 4.5.

The last two projects are larger-scale efforts and have made substantial progress towards the goal of a realistic, fully verified OS kernel. At the time of writing, they are expected to complete their proofs in 2008. They differ in approach and main focus, but they also share a large portion of their verification technology. We look at Verisoft and L4.verified in detail in Sect. 4.6 and Sect. 4.7 respectively.

The table does not include any currently commercially available OS kernels, because none have been formally verified to the degree discussed here. Three popular ones, Trusted Solaris, Windows NT, and SELinux (Red Hat Enterprise Linux 4.1) have been certified to Common Criteria EAL 4, but this level does not require any formal modelling and is not designed for systems deployed in potentially hostile situations. However, the security policies of SELinux have undergone formal analysis (Archer et al., 2003); in one instance using TAME (Archer et al., 2000; Archer, 2000, 2006), which is based on the PVS prover (Owre et al., 1996), and in another instance by a different group using model checking (Guttman et al., 2005). The security framework is based on the Flask architecture (Spencer et al., 1999) which was originally developed on the Fluke microkernel (Ford et al., 1996) and later ported by the NSA to Linux as the security architecture in SELinux. The two projects analysed the security policies themselves, but did not aim at proofs establishing that the the SELinux kernel correctly implements them. Greenhills' Integrity kernel (Greenhills Software, Inc., 2008) is currently undergoing EAL6+ certification, projected to complete in May 2008. EAL6 does involve formal modelling, but only semi-formal treatment of the high-level design and below. At this point this is all Greenhills seems to be aiming for. Of the projects in Fig. 5, only the seL4 kernel (Derrin et al., 2006) analysed in L4.verified is targeted at the commercial market.

4.2 Related Projects

A number of smaller projects are related to this overview. We do not survey these in depth, because they either targeted kernels less complex than microkernels or did not attempt proofs down to the implementation level. Nevertheless, they deserve mentioning in this context.

Flint The Flint project is not directly aimed at OS verification, but has made a number of important contributions in the area. In particular, Ni et al. (2007) describe a logic in which they verify low-level context switching code on the Intel x86 architecture how it typically would appear in an OS. Feng et al. (2008); Feng (2007) show how to verify low-level code in the presence of hardware interrupts and thread pre-emption.

MASK Martin et al. (2000; 2002) use the Specware tool (Kestrel Institute, 1998) to design and build the Mathematically Analyzed Separation Kernel (MASK). They provide a high-level information-flow guarantee in the sense of Sect. 3.2. The separation property is guaranteed by construction in their highest-level model, and by multiple formal refinement proofs down to a low-level design, which is close to an implementation. This low-level design is manually translated into C and reviewed against the Specware models. The kernel analysed is a separation kernel only, not a full microkernel. Code proofs as in the sections below were not attempted. Its main application was a cryptographic smartcard platform developed at Motorola. The project was a collaboration between the NSA, Motorola, and the Kestrel Institute.

Embedded Device Heitmeyer et al. (2006, 2008) report on the verification and Common Criteria certification of a software-based embedded device featuring a separation kernel. They do not go into detail about which device, which kernel, and which evaluation level exactly, but the report mentions 3,000 lines of C as the size of the kernel implementation, and data separation as its main purpose. The proof described in the report should qualify for EAL 7, but formal proof is of course only one of the requirements in a Common Criteria evaluation. Judging from its implementation size, the separation kernel is considerably less complex that a general purpose microkernel.

The verification approach is the TAME mentioned above (Timed Automata Modelling Environment) which was also used for analysing SELinux. The proof here, however, does aim to establish a formal relationship to the implementation. The verification comprises a Top Level Specification (TLS), a machine-checked proof that it satisfies the selected security policy, and a formal, but manual proof that the code implements the TLS. The TLS describing the abstract system is with 368 lines of TAME code very compact and the effort of mechanically verifying the separation policy on this model is given as only 2 weeks (Heitmeyer et al., 2006, Sect. 5.3). This roughly corresponds to the effort that the L4.verified project reports for its high-level security model (Elkaduwe, Klein and Elphinstone, 2007).

The separation policy of the embedded device kernel is divided into five parts: no-exfiltration, no-infiltration, temporal separation, control separation, and kernel integrity. The first two state that the running process cannot write data to other domains and that no other domains can write to the data of the current process. The term temporal separation is not used in the sense of timing or scheduling attacks, but means here that the separation kernel ensures that no sensitive data is left lying around at process switches. Control separation states that computation is strictly sequential: when one process computes, no other process is computing. Finally, kernel integrity means that user processes cannot change kernel memory.

The authors relate this model and proof to the C implementation by formal refinement (Abadi and Lamport, 1991), which they translate into Hoare-style pre/post condition proofs on the C code. The C code is annotated with assertions. They do not go into detail on which semantics or memory model of the C language is used in the verification, but the semantics includes at least pointers, potentially in a simplified form. As is common in refinement, program states of the C implementation are related to automata states of the TLS by a mapping function. We describe a similar approach in more detail in Sect. 4.3. To simplify verification, the code was divided into three categories: event code, trusted code, and other code. Event code directly corresponds to actions in the TLS, trusted code is code that potentially has access to secure information, and other code is code that is not directly security relevant. The first category is shown to implement the TLS actions by refinement (see Sect. 4.3), the second category is shown not to leak information between security domains, and the last category is only formally analysed where considered security relevant; for instance when it prepares access to the MMU. Two months were spent on these code conformance proofs (Heitmeyer et al., 2006, Sect. 5.4). The report admits that the *semantic distance between the abstract TLS required for a Common Criteria evaluation and a low-level C program is huge* (Heitmeyer et al., 2006, Sect. 6.2) and recommends investigation of a mechanical proof assistant to relate the code to the TLS. Comparing the time spent on code verification with the effort spent in the projects in Fig. 5 suggests that implementation correctness is indeed the hard part of OS verification. Although it investigates a less complex system and does not achieve the same level of assurance and mechanisation as the projects from the table are aiming for, the work of Heitmeyer et al. is an impressive demonstration that formal methods can be used cost effectively and successfully for OS verification.

AAMP7 The last project (Greve et al., 2004; Hardin et al., 2006) concerns the verification and Common Criteria EAL7 certification of the AAMP7 microprocessor (Rockwell Collins, Inc., 2003). This processor implements the functionality of a static separation kernel in hardware. Strictly speaking, this does not constitute kernel verification because an OS kernel is by definition software, but the policies and properties considered are very similar and closely related. The implementation language in this case is processor micro code, the prover used is ACL2 (Kaufmann et al., 2000). The functionality provided is again less complex than a general purpose microkernel, and the processor does not support dynamic delegation of authority or even online reconfiguration of separation domains. The processor does provide time and space partitioning in the sense of Sect. 3.2. The proof is based on a generic high-level model of separation kernels by Greve, Wilding and Vanfleet (2003), which after critique by Alves-Foss and Taylor (2004) was improved and generalised (Greve et al., 2005). The model is shown to implement partitioning. The proof also features a low-level design that exhibits a close one-to-one correspondence to the microprocessor code. The correspondence to code is not proven formally, but by manual inspection. In contrast to the previous project, the semantic gap here is very small and the low-level design is extensively validated by executing it in the ACL2 prover. High-speed execution of formal models is one of ACL2's distinguishing features and makes the validation of such low-level designs efficient and effective.

Instead of manually checking and cross-referencing the low-level design against the code, it would have been more satisfactory to translate the micro code into ACL2 automatically, thus closing the gap around manual translation completely. However, the level of detail provided in this project already seems to have surpassed what is usually expected for an EAL7 evaluation.

We now turn from the related projects to our survey of early work in OS verification.

4.3 Early Work

The table in Fig. 5 mentions three early OS verification projects: UCLA Secure Unix, PSOS, and KIT.

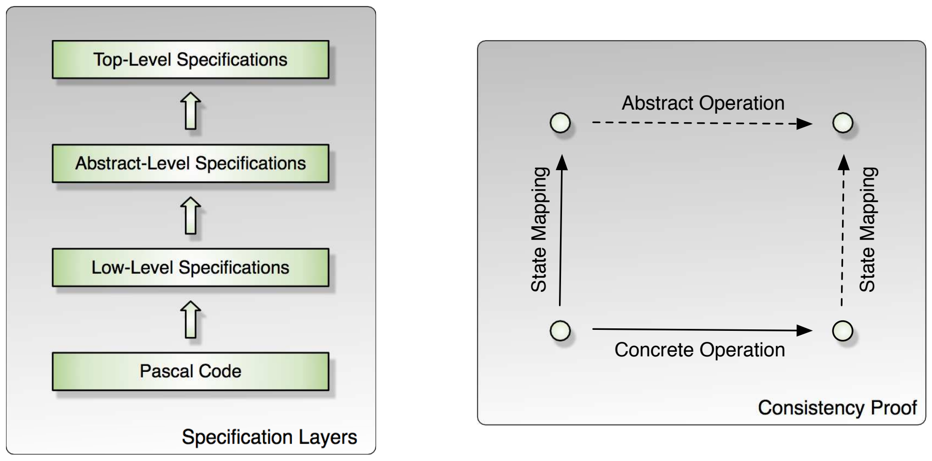


Fig. 6. UCLA Secure Unix. Specification layers and consistency proofs.

UCLA Secure Unix Walker et al. (1980) report on the specification and verification of UCLA Secure Data Unix, an operating system that was aimed at providing a standard Unix interface to applications, running on DEC PDP-11/45 computers. The verification effort was focused on the kernel of the OS, which by its feature description is close to the services of modern microkernels.[2] It provides threads, capabilities (access control), pages (virtual memory), and devices (input/output). The project was an early proponent of separating OS policy from kernel mechanism: It was argued that, building on formally verified kernel mechanisms, it would be significantly easier to prove properties about security policy enforcement (compared to implementing mechanism and policy as one). The kernel would only provide the mechanisms whereas policy would be implemented on the user level. The report does not state when exactly the project started, but it gives the total effort as about four to five person years (Walker et al., 1980, Sect. 4). According to Walker et al. (1980), first results appeared in 1977 (Popek et al., 1977); intermediate reports (Popek and Farber, 1978; Popek et al., 1979; Kemmerer, 1979) followed. As in other projects, the proof assumes that the compiler, hardware, and interactive verification tool work correctly.

Fig. 6 shows the verification approach taken by the project. The left hand side shows the specification layers that were used. At the bottom layer is the Pascal code of the kernel, which the XIVUS tool used in the project could read. The highest level is the top-level specification. In between, we have the low-level and abstract-level specifications. Note that this is not a layered software architecture as in Fig. 7, Sect. 3.2 where each level is implemented by different code. Instead, all these specification layers are formalisations of the same code artefact. They differ only in their level of abstraction. For example, the code provides full detail of everything; the low-level specification contains all variables that are retained between kernel calls, but uses more abstract structures for the rest; the abstract-level specification uses only abstract data structures such as lists and sets, but still describes all visible kernel operations; the top-level specification is a capability-based access control model of the kernel that abstracts away from all detail not important for this aspect. The intent was to show that the kernel provides a data-separation mechanism in the sense of Sect. 3.2.

[2] The first time microkernel-like concepts appear seems to be in the RC4000 Multiprogramming System in the 1960s (Hansen, 1970). The term microkernel was coined for the Mach kernel in 1988 (Rashid et al., 1989).

All of these specifications in UCLA Secure Unix are formally so-called state machines: they have a set of possible states, a current state, and a set of possible transitions between them. The proof then is to show that the specifications are consistent with each other, or formally, that the state machines simulate each other. This is easier to do in a number of smaller steps than in one big step, hence the multiple specification layers.

The right hand side of Fig. 6 shows the consistency proofs between the levels. Each such proof in the project proceeds by defining a function that maps program states of the concrete level to program states of the abstract level. For each operation in the concrete system, it is then shown that the corresponding operation in the abstract system transforms the mapped concrete state accordingly.

Although not called by that name at the time, the proof technique used in the project is *formal refinement* (Morgan, 1990; de Roever and Engelhardt, 1998), more specifically, data refinement. The idea is that a more concrete layer describes the same behaviour as an abstract layer, but with more detail. The abstract layer may say *pick an element of a set*, the more concrete layer may say *pick the smallest element of the set*, and the code may then implement the set as a sorted list and efficiently pick the head of the list, knowing that it will be the smallest element. The first refinement step in this example was reducing non-determinism (picking a specific element instead of any), the second level was pure data refinement (using a different data structure to achieve the same behaviour). The proof technique used in the project is an instance of the more general technique of forward simulation.

Walter et al. report that about 90% of these specifications were completed in the project as were about 20% of the code proofs. They managed to implement the kernel in a simplified version of Pascal—the XIVUS tool did not support pointers. Hardware access was modelled by isolating all functions that required such access. These functions were not covered by the standard Pascal semantics the tool supported, but apparently were axiomatised with pre/post conditions manually. The team did, however, extend the Pascal semantics with the ability to map certain global variables to hardware registers, which would have reduced the number of functions to be axiomatised. With regards to the proof, the report observes that invariants were an important component. Invariants here are properties that are true before and after each kernel call (but not necessarily during a kernel call). They are needed on each level to facilitate the consistency proof between levels and they often define the relationship between the different data structures in the kernel. The authors state that they were surprised by the degree of intrinsic relationship between data structures that was uncovered in these invariants and the specification of each layer. Data structures tended to be highly intermingled and not modular. This may be due to the nature of microkernels: Data structures that are fully modular and could be implemented strictly separately from the rest would be implemented separately and thus would not be part of the kernel any more.

The report concludes that the success in completing 90% of the specifications *was generally encouraging, but the proof effort sobering. Nevertheless, our specification work was very valuable, not only as a research effort, but because significant security errors were uncovered* (Walker et al., 1980, Sect. 2.5). This observation is repeated in other projects. While not providing the degree of assurance of a proof, a detailed specification effort already brings significant benefit in finding errors.

The code proofs are described as painful and tedious, and the report advises to wait for *more effective machine aids* (Walker et al., 1980, Sect. 3.3). It is the premise of the current OS verification projects that this situation has changed significantly in the last 30 years. The report also states that *Performance of the completed system is poor, an order of magnitude slower than standard Unix in some cases* (Walker et al., 1980, Sect. 4). The authors attribute this not to the fact that verification was performed, but to the non-optimising nature of their Pascal compiler and to the high context-switching cost that later microkernels were criticised for as well. As mentioned in Sect. 4, modern microkernels have overcome this limitation.

Regarding the development method, the authors find that the *recommended approach to program verification—developing the proof before or during software design and development—is often not practical* (Walker et al., 1980, Sect. 4), although they say that the system needs to be developed with verification in mind. The L4.verified project comes to a similar observation in Sect. 4.7.

A final interesting conclusion is that *The effort involved in specification and verification overwhelms that involved in coding, although it is less than the total amount spent in design, implementation, and debugging* (Walker et al., 1980, Sect. 4). Given that up to 70% of development effort is spent on testing and validation for software systems, verification seems to be a viable, if not cheaper alternative when it is applicable.

PSOS The provably secure operating system (PSOS) was a hardware/software co-design with the goal of *a useful general-purpose operating system with demonstrable security properties* (Neumann and Feiertag, 2003, Sect. 1). We use Neumann and Feiertag's readable, retrospective report from 2003 as the main source for this overview. The project started in 1973, delivered an early summary of its result in 1979 (Feiertag and Neumann, 1979), and its final report in 1980 (Neumann et al., 1980). Work on information flow analysis on PSOS continued until 1983 (Feiertag, 1980; Goguen and Meseguer, 1982, 1984) and in part lead to the development of SRI's PVS theorem prover. It was a larger and longer-term effort than UCLA Secure Unix; contemporary, but independent of it. Rather than focussing on the kernel only, PSOS considered the whole operating system, including new hardware design.

Considering that it involved new hardware, it is unclear how much of PSOS itself was implemented, but the design appears substantially complete. Code proofs were not undertaken. Neumann and Feiertag state that only *some simple illustrative proofs were carried out* (Neumann et al., 1980, Sect. 1). Nevertheless, PSOS is an impressive effort that pioneered a number of important concepts for OS verification and system design in general, and it aimed at applying formal methods throughout the whole implementation of the OS.

We have already mentioned the concept of hardware abstraction layers (HALs). Strong abstractions with small interfaces are a software engineering mechanism for building layered, complex systems. Ideally, the interface between layers n and $n + 1$ serves as the high-level, functional specification for level n, and at the same time as the enriched machine-model for the higher layer $n + 1$. Fig. 7 depicts such an architecture used in PSOS.

Not many such layered architectures exist, because they are hard to implement. A very widespread and successful example of a layered architecture is the TCP/IP network stack of an OS. It can comprise for example a hardware device driver at the bottom, an ethernet driver on the next level, an IP driver on top of that, followed by a TCP layer, and then application protocols.

The design of PSOS comprises 17 such layers, the bottom six of which were intended to be implemented by hardware. PSOS is not a kernel-based system (Feiertag and Neumann, 1979, p. 334). Instead it is based from the ground up on the principles of layer abstraction, modules, encapsulation and information hiding. It uses layer 0, i.e. tagged, hardware-enforced capabilities, as the pervasive mechanism for access control in the system. *Tagged* means that the hardware supports an additional bit indicating whether a word in memory stores a capability. These memory words may then only be accessed via special hardware instructions for capabilities and there are no instructions that can alter or forge a capability.

To design PSOS, the project initially developed the Hierarchical Development Method (HDM) (Robinson and Levitt, 1977) with its specification and assertion language SPECIAL. This language

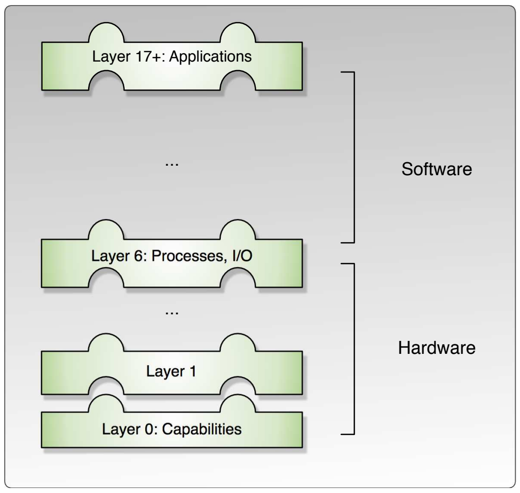


Fig. 7. PSOS layered system architecture.

was used to formally specify each module in the system. Each system layer could be comprised of a number of encapsulated modules with a formal interface. SPECIAL also allowed to specify mapping functions and abstract implementations relating modules between levels, which is already part of a potential implementation proof. Note that in such a layered system architecture, the layers each implement different, potentially additional functionality. Thus they are not just levels of specification abstraction as in Fig. 6. Level n is not necessarily invisible to every layer higher than $n + 1$. Nevertheless, the interface that layer n exports to higher layers can be used as its functional specification. This functional specification could independently have its own small stack of specification abstraction levels in the sense of Fig. 6.

Some of the specifications in PSOS ranged up into the application level, including a confined-subsystem manager, secure e-mail, and a simple relational database manager.

Neumann and Feiertag claim that the *PSOS architecture effectively dispels the popular myth that hierarchical structures must be inherently inefficient* (Neumann and Feiertag, 2003, Sect. 2.2). They make a valid point and argue this on a qualitative basis by looking at the design, but a quantitative comparison to a traditional system delivering the same services would be even more convincing.

Some of the underlying engineering principles that were used rigourously in PSOS have become mainstream techniques. Encapsulation and information hiding, for instance, are the basis of many object-oriented programming languages, and modules are a common concept in most languages. Strict layering as in PSOS is employed less frequently.

The design methodology of PSOS was later used for the Kernelized Secure Operating System (KSOS) (McCauley and Drongowski, 1979; Berson and Jr., 1979; Perrine et al., 1984) by Ford Aerospace. The Secure Ada Target (SAT) (Haigh and Young, 1987) and the LOgical Coprocessor Kernel (LOCK) (Saydjari et al., 1987) are also inspired by the PSOS design and methodology.

KIT Almost a decade after PSOS and UCLA Secure Unix, Bevier reports on the verification of KIT (Bevier, 1989*a*, 1987, 1988, 1989*b*), *a small operating system kernel written for a uni-processor computer with a simple von Neumann architecture* (Bevier, 1988, p. 1). KIT stands for *kernel for isolated tasks*, which is the main service that KIT provides. In addition, KIT provides access to asynchronous I/O devices, exception handling, and single-word message passing. It does not provide shared memory or virtual memory in the modern sense. It also does not offer the dynamic creation of processes or communication channels, or services such as file systems.

The implementation language is an artificial, but realistic assembler instruction set. With 620 lines of assembler source code and 300 lines of actual assembler instructions the kernel is extremely small and purposely very simple. It is several orders of magnitude smaller and less complex than modern microkernels.

KIT is significant because it is the first kernel that fully deserves the attribute *formally verified*. Despite its simplicity, it provides a useful service, formally verified down to realistic assembly source code. Bevier is the first to demonstrate conclusively that the level of detail required in OS implementation verification is not an intrinsic problem for formal verification.

The verification was conducted in the Boyer-Moore theorem prover (Boyer and Moore, 1988), the predecessor of the ACL2 prover that was also used in the verification of the AAMP7 microprocessor described in Sect. 4.1. The syntax of the prover has been criticised as hard to read because of its parenthesised prefix form that is derived from the programming language LISP. In fact, the Boyer-Moore logic itself is very similar to pure LISP. The logic provides no explicit quantifiers like $\forall$ and $\exists$. As mentioned before, one of the prover's major advantages is that the logic is efficiently executable. The prover provides a high degree of automation for an interactive system.

Very similar to UCLA Secure Unix and other refinement-based verifications, the proof of the KIT system shows correspondence between finite state machines which we explain below. In this case, there are three such finite state machines: the abstract, operational specification, the 'abstract kernel', and the kernel running on hardware.

The kernel running on hardware is realised as an operational semantics of the assembler instructions: The states of the state machine are the hardware machine states, including its memory, registers, flags and program counter; the transitions describe the fetch-execute cycle of the machine. The correspondence theorem is almost exactly the same as in Fig. 6 above. The restrictions of the Boyer-Moore logic make it necessary to reformulate it slightly to avoid an existential quantifier, but the content of the statement remains the same.

The medium level specification, i.e. the abstract kernel, *defines a scheduling algorithm for a fixed number of tasks, implements the communication primitives (including the delay of tasks which block on a communication), and handles communication with asynchronous devices* (Bevier, 1988, p. 9).

The abstract, top-level specification *defines the communication transitions in which a task may engage, but says nothing about how tasks are scheduled* (Bevier, 1988, p. 1). This abstract specification effectively provides a model of several communicating tasks running concurrently. The correspondence proof shows that the kernel correctly implements this abstraction on a single CPU.

There is no specific security model that KIT implements, but the top-level specification seems strong enough to at least imply data separation between processes.

Bevier and Smith later also produced a formalisation of the Mach microkernel (Bevier and Smith, 1993*a*,*b*). They did not proceed to implementations proofs, though.

After KIT, more than a decade passed without any larger-scale, serious attempts at formally verifying the implementation of an operating system. UCLA Secure Unix, PSOS and KIT had shown that it was possible in principle, and had pioneered some of the techniques that could be employed, but the systems that were implemented were either unrealistically small like KIT, or were slow and/or incompletely verified like PSOS and UCLA Secure Unix. Formal verification on the required scale seemed prohibitively expensive, and machine support not yet sufficiently advanced.

The first decade after 2000 now sees a renewed interest in this topic. The next sections survey these projects, starting with VFiasco.

4.4 VFiasco

The VFiasco (Verified Fiasco) project started in November 2001. Hohmuth et al. (2002*a*,*b*) detailed the main ideas and the approach of the project in 2002.

Fiasco (Hohmuth and Härtig, 2001) is a binary compatible re-implementation of the high-performance, second generation microkernel L4. The Fiasco implementation specifically emphasises reliability, maintainability, and real-time properties. In exchange for these, it sacrifices a small part of the performance the L4 kernel is known for. The implementation language is C++ with isolated interface code and optimised IPC in assembly. This choice addresses the maintainability goal of the project—early L4 implementations were written directly in hand-optimised assembly code.[3] The real-time focus expresses itself in the fact that almost all of the Fiasco code is fully preemptible. That is, apart from some short operations, it runs with hardware interrupts enabled. This leads to very short reaction times to external events (which are delivered via interrupts), but it makes the implementation and therefore the verification of the kernel considerably more complex.

The initial report (Hohmuth et al., 2002*a*) already recognises a number of issues that are crucial for the verification of modern microkernels: a precise formal semantics of the implementation language and the fact that kernel code runs under a much more complex view of memory than normal applications. The latter interferes with usual descriptions of programming language semantics. For example, most semantics textbooks (Winskel, 1993) assume that memory can be seen as a consecutive array of bytes and that memory reads and writes can never fail. None of these are true for kernel code. Since it is the code that implements the basis of the virtual memory abstraction, effects like page faults are fully visible to the code.

The proposed solution is to set up an invariant for well-behaved memory ranges. Traditional programming language semantics can be employed for code running only in those well-behaved ranges. This is expected to cover the largest percentage of the kernel code. The memory invariant might be broken temporarily by the kernel, for instance for MMU manipulations. For these cases a more complicated semantics must be used.

Next to its virtual memory model, the second area of contributions of the VFiasco project is its modelling of the C++ language for the verification of low-level code. The approach is to translate a C++ program directly into its semantics in the theorem prover PVS. This idea of a semantics compiler had previously been used in the LOOP project (Huisman and Jacobs, 2000) for Java. It avoids defining the syntax of C++ in the theorem prover and instead outputs functions that transform the program state and describe the operation of the C++ program. C++ is a very large and not a very clean language. It was not designed with verification in mind. Nevertheless, the project managed to produce a detailed model for C++ data types (Hohmuth and Tews, 2003). The authors use the under-specification present in the C++ standard to model type-safe fragments of the language. The C++ model also includes structurally unpleasant statements like goto. In an interesting case study, Tews (2004) demonstrates the verification of Duff's device, an abuse of C++ control flow features at whose discovery its inventor Tom Duff said he felt *a combination of pride and revulsion* (Duff, 1983). It is illustrative of the programming tricks some kernel programmers like to employ.

The last publication of the project in 2005 again summarises the approach on C++ verification (Hohmuth and Tews, 2005). The VFiasco approach to C++ source code verification was

[3] Later L4 implementations such as L4Ka::Pistachio (System Architecture Group, 2003) were written in C++ as well, retaining their high performance characteristics.

continued in the Robin project (Tews, 2007; Tews, Weber and Völp, 2008) which investigated the verification of the Nova hypervisor, a different microkernel aimed at OS virtualisation, running multiple guest OS instances such as Linux on top. Like Fiasco, Nova is an L4-based kernel. Robin completed in April 2008 and produced about 70% of a high-level specification for the Nova kernel (Tews, Weber, Poll, van Eekelen and van Rossum, 2008). Neither VFiasco not Robin did manage to verify significant portions of the implementation.

4.5 EROS/Coyotos

The Coyotos kernel (Shapiro et al., 2004) is the successor of the capability-based microkernel EROS (Extremely Reliable Operating System) (Shapiro et al., 1999), which itself is the successor of the KeyKOS (Hardy, 1985) kernel.

Like L4, the EROS system is a second-generation microkernel. Its emphasis is on solid performance and building secure systems. Shapiro and Weber (2000) formalised and analysed its security model in a pen-and-paper proof. The model was based on the take-grant model of capability distribution (Lipton and Snyder, 1977) where, apart from invoking a capability to read or write data, one may *grant* a capability to someone else, or one may have the right to *take* a capability from someone else. Both these rights are themselves conferred via capabilities. Additionally, new capabilities can be created in the model, and existing ones deleted. Lipton and Snyder (1977) originally analysed this take-grant model and observed that it is possible for the OS to decide efficiently for each of these operations whether the operation would allow a subject to break a global security policy at some point in the future. The model can therefore be used to implement mandatory access control.

The security model of the EROS kernel was not formally connected to the implementation. This was supposed to change for the successor kernel Coyotos. Shapiro et al. (2004) laid out a plan for the design and formal implementation of the new kernel. One of the central features of this approach was to create a new programming language for kernel implementation first—one that is safe, clean, and suitable for verification and low-level implementation at the same time. The idea was not necessarily to invent new language features, but rather to pick and choose from existing research and combine the necessary features into a targeted language for implementing verified OS kernels (Sridhar and Shapiro, 2006; Shapiro, 2006). This is a far-reaching and long-term effort, but one with great appeal: The main kernel languages assembler, C, and C++ are old, with many unsafe features and were not built to ease program verification. A new, modern language suitable for kernel implementation and verification might take a long time to design and get right, but would certainly benefit future verification efforts.

The Coyotos project has made significant progress on the design and implementation of the kernel itself (Shapiro, 2008), as well as on the design of BitC, the proposed new implementation language. Sridhar and Shapiro (2006) describe the language specification and a number of small innovations in its type system. The language is designed to be type safe and fast, and to allow reasoning about the low-level layout of data structures. The latter is important for issues such as predictable cache performance and hardware interfaces. Its LISP-like syntax is reminiscent of the ACL2 prover. Some early experiments were conducted, but the project has not yet published any kernel verification results or a logic for reasoning about BitC programs.

4.6 Verisoft

The Verisoft project is a large-scale effort to demonstrate the pervasive formal verification (Bevier et al., 1989) of a whole computer system from the hardware up to application software. The project

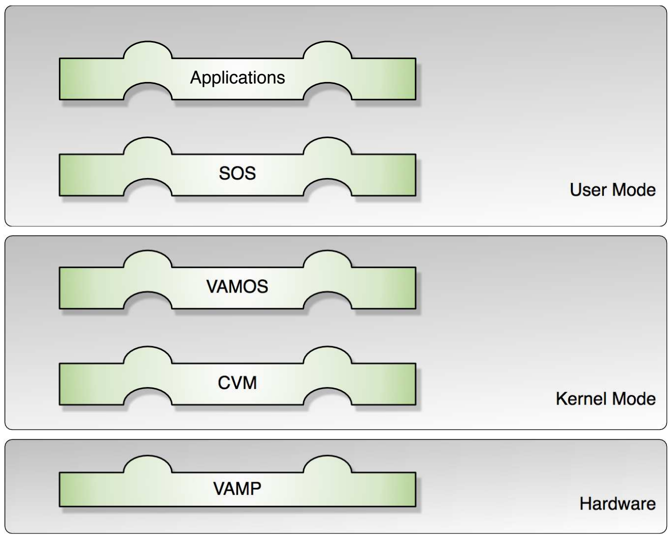


Fig. 8. Verisoft: system layers.

started in 2003 and received EUR 14.8 million in funding from the German government (Verisoft Project, 2006) for a period of four years until its scheduled end in 2007. The successor project Verisoft XT began immediatly afterward in 2007. Pervasive verification in this context means that the project does not plan to rely on the correctness of the compiler or even the instruction set model. Instead, all of these steps are to be formally verified as well, such that there is a complete, unbroken formal chain from hardware to applications.

Hillebrand and Paul (2008) give an overview of the verification technology and approach. The project consists of a number of parts, some of which involve confidential work for industry partners. We concentrate here on the public part which contains most of the OS work. According to project manager Tom in der Rieden, the total effort spent on this part is estimated to be 30 person years.

Fig. 8 shows the verification approach. Similar to PSOS, it is a layered approach with hardware at the bottom and application software at the top. In contrast to PSOS, the focus here is not on a design methodology. Work has progressed far beyond the design stage, all of the artefacts are implemented and formally specified. Most of them are verified in pen-and-paper proofs, and roughly 75% of the proofs are machine-checked in the theorem prover Isabelle/HOL (Nipkow et al., 2002) at the time of writing. The parts we refer to as *verified* below are those that have been verified in Isabelle/HOL.

As mentioned, the bottom layer of Fig. 8 is the hardware, in this case the VAMP microprocessor, a variant of the DLX architecture (Hennessy and Patterson, 1996). Its instruction set behaviour has been formally verified down to the gate level (Beyer et al., 2006; Dalinger et al., 2005) in the theorem prover PVS, in part before the Verisoft project started. The instruction level specification has been ported to Isabelle which was used for the OS verification part of the project. The VAMP processor is a real processor in the sense that it exists in silicon, but it is not widely used (commercially or otherwise). It supports simple address translation, memory mapped I/O devices (Hillebrand et al., 2005; Alkassar et al., 2007) and a pipeline architecture. It provides a firm, formal basis for the software stack on top.

The next layer in Fig. 8 is the CVM (Communicating Virtual Machines) layer. It comprises the hardware-dependent part of the OS kernel and establishes a hardware independent interface for the rest of the kernel to run on (in der Rieden and Tsyban, 2008). This division is convenient for the verification effort, because it isolates those parts of the kernel that involve assembly-level constructs. They have to be verified in a more detailed model than the rest of the system and are consequently more labour intensive. This software layer is implemented in C0A, a variant of the C0 language with inline assembly. We will discuss C0 below in more detail. The CVM layer verification has been mostly completed. Proofs for some of the device drivers are missing, but a hard disk driver has been verified as part of the memory paging mechanism (Alkassar et al., 2008; in der Rieden and Tsyban, 2008).

The next layer up in Fig. 8, VAMOS (Dörrenbächer, 2006), delineates code running in the privileged mode of the hardware. Together with the CVM layer, it makes up the OS kernel. It is implemented purely in the C0 programming language. The VAMOS/CVM combination cannot be called a microkernel in the strict sense, because it includes a kernel-mode device driver and a (verified) memory paging and disk swapping policy (Alkassar et al., 2008). In a true microkernel, this functionality and policy would be implemented in user mode. The idea is close, though, and including this driver and pager in the kernel considerably simplifies the memory view of the processes running on top of the kernel. If all components are verified, the question is not one of reliability any more, but one of flexibility and performance only. Verification of the VAMOS layer has progressed substantially (Starostin and Tsyban, 2008), but not as far as the CVM layer. At the time of writing, a number of small kernel calls are missing. The hardest part, IPC, is 90% complete.

On top of the VAMOS/CVM kernel, a simple operating system (SOS) is implemented in the user-mode of the hardware. The SOS runs as a privileged user process, that is, it has more direct access rights to the hardware than what is granted to usual applications. It provides file based input/output, IPC, sockets and remote procedure calls to the application level (Hillebrand and Paul, 2008, p. 156). This level as well as the next is implemented in C0. SOS has currently only been verified in part.

The uppermost layer in Fig. 8 is the application layer, where a number of example applications have been implemented and in part formally verified, including a small email client (Beuster et al., 2006).

Fig. 8 depicts only those artefacts that are implementations in the traditional sense. They are written in assembly or the C0A and C0 languages. If we introduce specifications, the picture becomes more complex, gaining a number of layers. For instance, the CVM layer has an abstract specification that in conjunction with the instruction set model of the hardware is used as a basis for specifying and implementing the hardware independent layer VAMOS. The CVM layer itself also has a C0A implementation that formally refines the abstract CVM specification (in der Rieden and Tsyban, 2008). If we view specifications as generalised programs, we get (from the bottom): hardware, CVM implementation, CVM specification, VAMOS. Each of the layers in Fig. 8 has at least one such implementation and specification part.

The goal of the project was to demonstrate pervasive verification, i.e. to end with one final, machine-checked theorem on the whole system, including devices. This theorem for example should state the correct operation of the compiled email client on VAMP gate level hardware. This in turn means that we cannot stop at source code implementations. For each of the layers in Fig. 8, we not only have to consider their C0 implementation, but we also need to relate this implementation to compiled VAMP code. As for the relationships between the other layers, this could in theory be achieved by the same kind of manual simulation proof, but since the transformation from source to assembly implementation is automated by the compiler, it is far more economic to verify the correctness of the compiler instead. The compiler correctness theorem then merely needs to be strong enough to imply the inter-layer proof that would otherwise have been done manually.

Consequently, compiler verification was a strong focus in the Verisoft project. Starting with an initial report in 2005 by Leinenbach et al. (2005), the project has made significant progress towards a fully verified, non-optimising C0 compiler that supports mixing inline assembly with C0 code (Leinenbach and Petrova, 2008). In contrast to other efforts (Leroy, 2006; Blazy et al., 2006; Klein and Nipkow, 2006), this not only includes a high-level executable compiler written in the theorem prover's specification language, but also a C0 implementation of this high-level specification (Petrova, 2007). Due to time constraints the front-end of the compiler (parsing the syntax of C0) was excluded and remains unverified.

There are three languages involved in the compiler verification effort: C0, C0A, and VAMP assembler. The top-level C0 language used in the project is C-like in its syntax and Pascal-like in its semantics. It is a clean, formally well-defined, type-safe programming language. It does not contain arbitrary pointer arithmetic or unsafe pointer casts. In its pure form, it is not suited to fully implement a low-level OS kernel. For those parts of the system that require direct hardware access, the language is enriched with the possibility to mix inline VAMP assembly code with C0, arriving at the C0A language. The VAMP assembler finally is the instruction set provided by the hardware platform.

The compiler correctness theorem (Leinenbach and Petrova, 2008) states that the compiled assembler program simulates the semantics of the high-level C0 or C0A program. Leinenbach and Petrova (Leinenbach and Petrova, 2008, Sect. 4.4) also briefly consider the bootstrap problem (Goerigk and Hoffmann, 1998): how to arrive at a trusted, verified compiler binary when the compiler is used to compile itself. They solve this by two external means: code generated by the theorem prover out of the high-level specification, and translation validation for the initial compilation. The authors argue that it is extremely unlikely that the same error remains unspotted by both techniques at the same time on a formally analysed implementation. There is another way to do this with fully formal guarantee: verify the initial compilation, using the existing VAMP instruction set semantics (either manually or automatically). This was not attempted. Leinenbach and Petrova give the size of the compiler implementation with 1,500 lines of C0 code and 85,000 lines of Isabelle proof (Leinenbach and Petrova, 2008, Sect. 5).

The main code verification technology used in this project as well as in the L4.verified project below was developed by Schirmer (2004, 2006). The tool is a generic environment in the theorem prover Isabelle for the verification of sequential, imperative programs that can be instantiated to a number of different languages, in this case C0 and C0A. The tool set includes a Floyd-Hoare-style logic (Hoare, 1969) for program verification, a big-step semantics that is convenient for language proofs such as compiler correctness and type safety, and a small-step semantics that is detailed enough to allow reasoning about interleaved, concurrent execution and non-terminating programs. These semantic levels are connected to each other by equivalence proofs. They are used at their appropriate level of detail in the simulation proofs between the software layers of the project. The verification environment also integrates with tools such as software model checkers (Daum et al., 2005) that can automatically discharge certain kinds of proof obligations, thereby reducing the manual proof effort.

Apart from work directly on the system layers explained above, the project also verified and published work on a number of libraries, including a string library (Starostin, 2006) and a big-integer library (Fischer, 2008). As in the rest of the project, the focus was on ease of verification, not on providing a high-performance implementation.[4] Most of the artefacts and proofs described here, including the compiler, are publicly available (Verisoft Project, 2008).

Putting all these layers back together with simulation theorems, connecting gates to VAMP instructions, VAMP instructions to C0 implementations, C0 implementations to their next layer specifications and so on, it is in principle possible to arrive at the pervasive theorem that was the original goal of the project. Verisoft has not published this final theorem yet, but has come close. Work on the remaining OS kernel primitives is still in progress.

From the perspective of the OS verification overview in this paper, the Verisoft project focused on pure implementation correctness only and did not investigate high-level security policies or access control models of the OS in the sense of Sect. 3.2. From the projects surveyed here, it clearly demonstrates the most comprehensive and detailed implementation correctness statement.

Given this achievement, it is a shame that the resulting verified system is not likely to see widespread use and through it have a direct impact on deployed systems in the short term. The system is available for the VAMP processor only, and since performance was not a focus, is unlikely to convince application programmers by its verified merits alone. There is no indication that this is due to a problem of the technology that was developed; rather it is the consequence of where the focus of the project was. Verisoft showed conclusively in an impressive verification and engineering effort that a fully trustworthy base for computing systems is firmly within reach of current technology.

4.7 L4.verified/seL4

The L4 verification effort includes two different, but closely related projects: seL4 and L4.verified. The L4.verified project aims to provide a machine-checked, formal correctness proof of a high-performance implementation of the seL4 microkernel. The seL4 (secure embedded L4) kernel (Elphinstone et al., 2007) is an evolution of the L4 microkernel (Liedtke, 1995) with emphasis on efficient support for security and embedded systems. The author of this overview is affiliated with the L4.verified project.

After a small pilot project in 2004, L4.verified started in 2005 (Tuch et al., 2005), concurrently with the seL4 project whose task it was to design and implement seL4. The stated goal of these two projects was to provide an implementation correctness proof for seL4 with the kernel running on a mainstream embedded processor within 10% of the performance of L4, currently the fastest microkernel in the world. The initial pilot project resulted in a case-study on virtual memory of the existing L4 kernel (Tuch and Klein, 2004; Klein and Tuch, 2004), a high-level specification of L4 IPC (Kolanski and Klein, 2006) and initial design ideas for seL4 (Elphinstone, 2004).

The seL4 project was concluded successfully by the end of 2007 (Elphinstone et al., 2007). The resulting seL4 design provides the following kernel services: threads, IPC, virtual memory control, capabilities, and interrupt control. The design was implemented in C and assembly on the ARM11 platform (ARM, 2000). Its performance matches that of L4; in some cases that are important for virtualisation, it even exceeds that of L4 (Elkaduwe et al., 2008).

The capability system of seL4 is similar to that of the EROS kernel (Shapiro et al., 1999) mentioned above. Like the latter, it is based on the take-grant model (Lipton and Snyder, 1977). One particular feature of seL4 is that there is no implicit kernel resource consumption for user processes—capabilities explicitly manage and account for all resource aspects (Elkaduwe, Derrin and Elphinstone, 2007; Elkaduwe et al., 2008). This is significant because the old L4 kernel is

[4] Wolfgang Paul, the scientific director of the project, mentioned at the SSV'08 workshop in Sydney, Australia that a factor of 100 is considered to be an acceptable slow-down for easing the verification task.

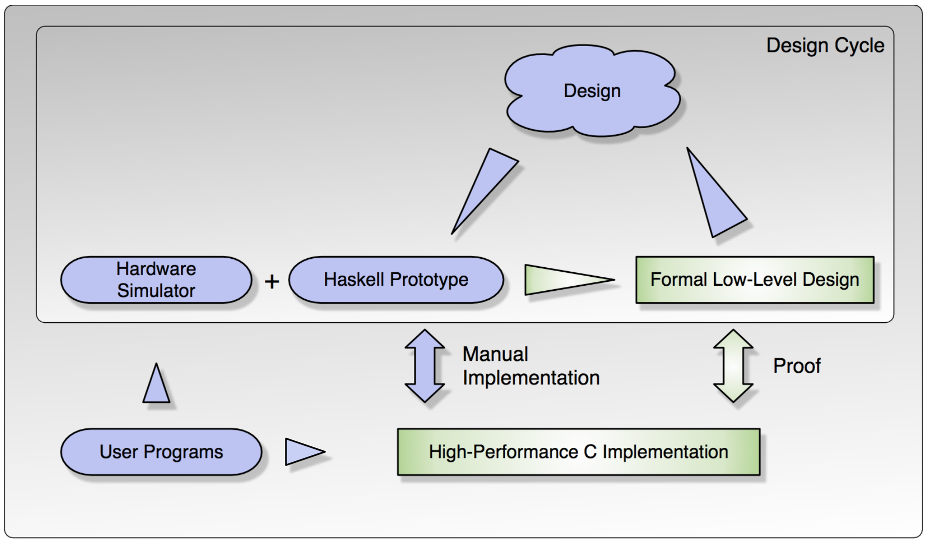


Fig. 9. The seL4 design and rapid prototyping method.

vulnerable to a simple denial of service attack: A thread can exhaust kernel memory by constantly creating new threads. EROS solves this problem by viewing kernel memory as a cache of the system state (Shapiro et al., 1999). This solution makes it hard to predict worst-case execution times and is therefore not well suited for embedded and real-time applications. Separation kernels solve the problem by giving each separation domain a fixed quota of kernel memory. This solution is not flexible enough for a full microkernel design. The seL4 design on the other hand provides a solution well suited for embedded and real-time systems.

Capabilities in seL4 are not hardware-enforced like in PSOS. Instead they are software-implemented on a standard processor architecture and are kernel-enforced. They provide fully dynamic, flexible mechanisms for hierarchically delegatable authority.

An important factor in the success of the seL4 kernel design was the tight integration of the two teams: While the design was mainly driven by the NICTA OS group in the seL4 project, the concurrent verification effort in L4.verified provided continuous, early feedback that was taken into account by the design group. This way, a good balance between external performance requirements and verification constraints was maintained. The tight integration kept the focus on performance and realistic architecture decisions instead of trading off more ease-of-verification for slower kernel implementation as in Verisoft. The verification influence kept the design to a small, manageable size instead of the greater complexity of a system such as PSOS. The seL4 and L4.verified projects jointly developed a methodology that facilitated this tight integration between formalisation and kernel prototyping. Fig. 9 gives a summary of the idea.

The most important activities in designing a microkernel are discussions between the developers, e.g. on a whiteboard. To validate the design decisions, in the traditional OS approach a prototype kernel is implemented in C or similar. This exposes performance and data structure layout problems and allows direct testing of applications on top of the kernel. However, this prototyping has a long overhead time between new design ideas and new prototype versions (on the order of weeks and months). Additionally, initial case studies in the L4.verified project (Tuch et al., 2005) and the experience in VFiasco showed that starting the verification directly from the C source without any higher-level specification should be expected to be a difficult and long process.

In contrast to the OS approach, the traditional formal methods approach would take the design ideas, formalise them into a high-level specification first and then analyse that specification. This has the advantage of exposing conceptual problems early. The disadvantages lie in not exercising the design through user applications and not exposing performance and data structure layout problems.

In the seL4 approach, the prototype is not written in C, but in a high-level, functional programming language that is efficiently executable and close to the notation of the theorem prover. In this case, the language chosen by the OS group was Haskell (Jones, 2003), a pure, side-effect free functional language that the kernel implementers felt comfortable with. Given a hardware simulator for the target platform that transforms hardware kernel trap instructions into calls to the Haskell prototype, normal user application binaries can still exercise the design directly. The Haskell prototype can be made sufficiently low-level to expose data layout problems immediately. Performance is only evaluated in a qualitative way, by observing the number of system calls necessary to achieve a desired goal and by inspecting the Haskell code of these calls to judge how well they could be implemented in C. Since Haskell is very close to Isabelle/HOL, it can be automatically translated into the theorem prover. In this way, we arrive automatically at a precise, formal low-level design which is the basis of further formal analysis. Both the formal analysis and the validation against user binaries are able to feed back rapidly into the design cycle. Changes in the design can be implemented in a manner of days, sometimes hours (Heiser et al., 2007; Elphinstone et al., 2007, 2006).

For the purpose of running the Haskell kernel prototype, the two projects jointly developed a hardware simulator generator that takes an instruction set specification together with a simple device description and turns it into an efficient instruction-level simulator as well as an Isabelle/HOL formalisation. This hardware formalisation can then form the basis for assembly level verification. By running the simulator, the formalisation can be extensively validated against the real processor. Trust is reduced from a manual formalisation to the correct operation of the generator.

The total effort spent on designing, implementing and validating the seL4 kernel comes to 6 person years. This includes benchmarking and porting the L4/Linux (Leslie, van Schaik and Heiser, 2005) and Iguana (NICTA, 2008) operating systems to the new kernel.

We now turn from the seL4 design to its verification in the L4.verified project. Fig. 10 gives an overview of how the design fits into the verification picture. The L4.verified refinement approach looks similar to that of UCLA Secure Unix. On the bottom layer, we find the source code implementation of the kernel in C and assembly. To reason about the implementation, this lowest model also needs to include some detail about the hardware such as a model of the MMU, exceptions, and interrupts. The next layer up is the low-level design—the Isabelle/HOL translation of the Haskell prototype mentioned above. It contains implementations of all important data structures with a high degree of low-level detail like the number of bits that are used to store capabilities, doubly-linked lists implemented by pointers etc. The second highest layer in Fig. 10 is the high-level design. It contains a description of all user-visible kernel operations, but leaves out data structures and detail where possible. For instance, the scheduler on the abstract level is specified simply by saying *pick any thread in the system that is runnable*, but in the low-level design, a data structure is provided for tracking which threads are runnable and which priority they have. The top level of Fig. 10 is the access control model of seL4 (Elkaduwe, Klein and Elphinstone, 2007). It abstracts strongly from detail and only specifies how capabilities are distributed in the system. The security property proven in L4.verified states that the kernel can effectively isolate subsystems and can enforce flexible global security policies.

All specifications in Fig. 10 have been completed (Cock et al., 2008). The security specification is ca 300 loc, the high-level design 3,000 loc, the low-level design 7,500 loc, and the C code

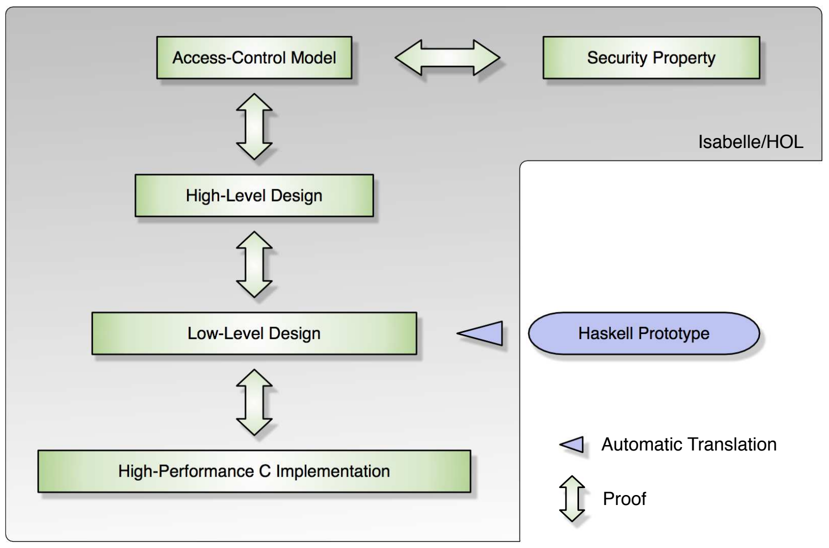


Fig. 10. L4.verified: refinement layers.

10,000 loc. The project is concurrently working on a formal refinement proof between the access control model and this high-level design and also on the refinement proof between the low-level design and the C implementation (Elphinstone et al., 2007, 2006). The project has completed the machine-checked proof that the security property holds on the access control model (Elkaduwe, Klein and Elphinstone, 2007); this took 2 person months. The project has also completed the machine-checked proof in Isabelle/HOL that the Haskell prototype providing the low-level design of seL4 correctly implements its high-level specification (Cock et al., 2008). This fills one of the largest semantic gaps in the chain of Fig. 10. The refinement proof between abstract and executable specification is 100,000 lines of Isabelle proof. It took five person years.

The large-scale proof between high- and low-level design produced a number of innovations in proof engineering (Cock et al., 2008). They allow multiple people to work with a high degree of independence on large-scale proofs for refinement of functional programs, thus reducing the overall time required for such verifications.

The main approach for C-level implementation verification in L4.verified is to take not a slightly changed variant of C, such as C0, but to formally treat a realistic, true subset of C. This way, the implementation can be compiled with standard tools and used directly for commercial systems on mainstream platforms. The compiler is part of the trusted tool chain this project. As Verisoft has shown, this gap can in principle be eliminated as well.

The subset of C the project treats is strictly pragmatic. It includes unsafe C features that the OS team considered mandatory for the efficient implementation of the kernel. Among these features are arbitrary pointer arithmetic and unchecked type casts. As mentioned previously, L4.verified uses Schirmer's verification environment (Schirmer, 2006) developed in the Verisoft project. For this, some features of C were restricted: taking the address of a local variable is not allowed; neither is the use of unions and function pointers. These three could be added to the model with some effort, but they were not needed for the implementation of seL4. The model does not include inline assembly. Instead, assembly portions are isolated into separate functions, which are verified separately against their specification. This specification is then used in the rest of the C program.

The C program state may be enriched with hardware state that is only accessible to these assembly functions. The technique is similar to Verisoft's XCalls (Leinenbach and Petrova, 2008).

One salient feature of the L4.verified C model is that it is formalised at a very detailed level to support reasoning about unsafe language constructs. For example, it uses a formalisation of fixed-size machine words (Dawson, 2007) such as 32-bit or 64-bit words, instead of assuming general natural numbers. At the same time, the formalisation provides proof techniques and abstractions that allow efficient, abstract verification of program parts that do not use unsafe C features (Tuch and Klein, 2005; Tuch et al., 2007; Tuch, 2008*b*). Tuch's (2008*a*) work proves two high-level reasoning techniques correct and integrates them into the C verification framework: separation logic (Reynolds, 2002) and multiple typed heaps (Bornat, 2000; Burstall, 1972). The first is a logic for pointer-based programs that is currently gaining popularity in the verification community. The second allows the program heap to be cut into separate, non-interfering parts by type to simplify verification.

The C code is parsed directly and automatically into the theorem prover. The technology has been tested in a complex case study of the L4 kernel memory allocator functions (Tuch et al., 2007). As part of the project, Kolanski (2008) has developed an adaptation of separation logic that allows reasoning about the virtual memory layout of the machine. This logic can be used to justify a simpler memory model for the majority of the code, and to reason directly about MMU manipulations.

The total effort spent on all tool, library, logic, and C model development spent in the project so far was an estimated 10 person years.

Coming back to the already completed, large-scale proof between high-level and low-level design, the formal implementation correctness statement required the project to show a large number of internal consistency invariants on both models. These invariant proofs made the strongest direct contribution to the seL4 design process. One of the simplest of these invariants states that all references in the kernel, including all capabilities, always point to valid objects of the right type (i.e. thread capabilities always point to thread control blocks). This is easy to establish for most kernel operations, but becomes interesting in the case of the *retype* operation which allows processes to re-use memory, destroying old kernel data structures and initialising new ones in their place. The kernel implements a number of complex bookkeeping data structures and enforces restrictions on the retyping operation. These elaborate data structures and restrictions are collectively designed to make sure that the internal consistency cannot be broken by cleverly misusing the operation alone or in combination with any other kernel operation.

The invariant proof shows that this design and interplay between bookkeeping and API restrictions indeed works: No user will be able to create unsafe pointers inside the kernel, spoof capability derivations, or gain elevated access to kernel resources through obscure corner conditions or wrong parameter encodings. The project has established these consistency invariants for all kernel operations, for all possible inputs, and for all possible sequences of kernel invocations. Cock et al. (2008) describe some of them in more detail. Even without C-level implementation verification yet, this proof constitutes a strong validation of the seL4 design and the mechanisms it provides.

The L4.verified project is scheduled to complete by the end of 2008. Introduction of seL4 or a derivative of it into the marketplace is currently being planned with Open Kernel Labs, Inc. (2007).

5 Conclusion

In this article, we have given a brief overview on software verification in general and operating systems verification in particular. We have looked at modern microkernels and the security policies they enforce. Finally, we have surveyed the state of the art in OS verification, looking at nine different projects in the area from 1973 to 2008.

There is currently still no general-purpose OS or OS kernel that would deserve the tag “fully formally verified”. There are a number of smaller systems that have undergone EAL7 certification, but none of the systems have managed to close the gap so far that manual translation is not needed between the code running on the machine and the code verified in the theorem prover. The AAMP7 project came close to this goal. The Verisoft project could close the gap completely and go beyond source code verification, but the official project has terminated; work on the VAMOS kernel verification is progressing on a smaller scale only. The L4.verified project includes source code verification and is still ongoing. It is scheduled to complete by the end of 2008. Both, Verisoft and L4.verified have a realistic chance at creating a fully verified OS kernel by the end of 2008: one with a completely pervasive verification approach, arguably on a niche architecture, the other with source code verification and a full high-performance implementation on commercial systems.

The projects surveyed demonstrate that OS verification is still a hard and expensive task, but in contrast to 30 years ago, it is clearly within the reach of current technology. Proof automation, memory and machine models, language semantics, prover user interfaces, proof libraries, and program logics have developed significantly in those 30 years. Instead of on the order of hundred lines of assembly code in KIT, we can now formally verify thousands of lines of complex C code. Keeping the verified code base updated and verified, repeating the verification effort, or adapting it to slightly larger systems should be faster and cheaper: a significant part of the effort in existing projects was spent on the further development of verification tools, on formal models for low-level programming languages and paradigms, and on general proof libraries. The sharing of substantial parts of the verification tools between Verisoft and L4.verified demonstrates that there is a significant degree of re-usability in these developments and large parts have been made publicly available. Future efforts will be able to build on these tools and reach far-ranging verification goals faster, better, and cheaper.

Acknowledgements The author would like to thank Elisabeth Meister, Timothy Bourke, Mark Hillebrand, Tom in der Rieden, Norbert Schirmer, and Hendrik Tews for their feedback and for reading drafts of this article.

References

Abadi, M. and Lamport, L. (1991), ‘The existence of refinement mappings’, *Theoretical Computer Science* **82**(2), 253–284.

Abrial, J.-R. (1996), *The B Book—Assigning Programs to Meanings*, Cambridge University Press.

Alkassar, E., Hillebrand, M., Rusev, S. K. R. and Tverdyshev, S. (2007), Formal device and programming model for a serial interface, *in* B. Beckert, ed., ‘Proceedings, 4th International Verification Workshop (VERIFY)’, Vol. 259 of *CEUR Workshop Proceedings*, Bremen, Germany, pp. 4–20.

Alkassar, E., Schirmer, N. and Starostin, A. (2008), Formal pervasive verification of a paging mechanism, *in* C. R. Ramakrishnan and J. Rehof, eds, ‘Tools and Algorithms for the Construction and Analysis of Systems (TACAS)’, Vol. 4963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 109–123.

Alves-Foss, J. and Taylor, C. (2004), An analysis of the gwv security policy, *in* ‘Fifth International Workshop on the ACL2 Prover and its Applications (ACL2-2004)’. Available from `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/alves-foss-taylor/ACL2004-final.pdf`.

Archer, M. (2000), ‘TAME: Using PVS strategies for special-purpose theorem proving’, *Annals of Mathematics and Artificial Intelligence* **29**(1-4), 139–181.

Archer, M. (2006), Basing a modeling environment on a general purpose theorem prover, Technical Report NRL/MR/5546–06-8952, NRL, Washington, DC, USA.

Archer, M., Heitmeyer, C. L. and Riccobene, E. (2000), Using TAME to prove invariants of automata models: Two case studies, *in* 'FMSP '00: Proceedings of the 3rd Workshop on Formal Methods in Software Practice', ACM, New York, NY, USA, pp. 25–36.

Archer, M., Leonard, E. and Pradella, M. (2003), Analyzing security-enhanced Linux policy specifications, *in* 'POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks', IEEE Computer Society, Washington, DC, USA, pp. 158–169.

ARM (2000), *ARM Architecture Reference Manual*.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K. and Ustuner, A. (2006), Thorough static analysis of device drivers, *in* 'EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006', ACM, New York, NY, USA, pp. 73–85.

Bell, D. E. and LaPadula, L. J. (1973), Secure computer systems: Mathematical foundations, vol. i, Technical Report MTR2547, The MITRE Corporation, Bedford, MA, USA. (ESDTR73278I).

Berson, T. and Jr., G. B. (1979), KSOS: Development methodology for a secure operating system, *in* 'AFIPS Conference Proceedings, 1979 National Computer Conference', Vol. 48, AFIPS Press, pp. 365–371.

Bertot, Y. and Castran, P. (2004), *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer-Verlag.

Beuster, G., Henrich, N. and Wagner, M. (2006), Real world verification—experiences from the Verisoft email client, *in* G. Sutcliffe, R. Schmidt and S. Schulz, eds, 'Proc. ESCoR 2006', Vol. 192 of *CEUR Workshop Proceedings*, pp. 112–115.

Bevier, W. R. (1987), A verified operating system kernel, Technical Report 11, Computational Logic Inc., Austin, TX, USA.

Bevier, W. R. (1988), Kit: A study in operating system verification, Technical Report 28, Computational Logic Inc., Austin, TX, USA.

Bevier, W. R. (1989*a*), 'Kit: A study in operating system verification', *IEEE Transactions on Software Engineering* **15**(11), 1382–1396.

Bevier, W. R. (1989*b*), 'Kit and the short stack', *Journal of Automated Reasoning* **5**(4), 519–530.

Bevier, W. R., Hunt, W. A., Moore, J. S. and Young, W. D. (1989), 'An approach to systems verification', *Journal of Automated Reasoning* **5**(4), 411–428.

Bevier, W. R. and Smith, L. (1993*a*), A mathematical model of the Mach kernel: Atomic actions and locks, Technical Report 89, Computational Logic Inc., Austin, TX, USA.

Bevier, W. R. and Smith, L. (1993*b*), A mathematical model of the Mach kernel: Entities and relations, Technical Report 88, Computational Logic Inc., Austin, TX, USA.

Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D. and Paul, W. J. (2006), 'Putting it all together—formal verification of the VAMP', *International Journal on Software Tools for Technology Transfer (STTT)* **8**(4), 411–430.

Blazy, S., Dargaye, Z. and Leroy, X. (2006), Formal verification of a C compiler front-end, *in* J. Misra, T. Nipkow and E. Sekerinski, eds, 'FM 2006: 14th International Symposium on Formal Methods', Vol. 4085 of *Lecture Notes in Computer Science*, pp. 460–475.

Bornat, R. (2000), Proving pointer programs in Hoare Logic, *in* R. Backhouse and J. Oliveira, eds, 'Mathematics of Program Construction (MPC 2000)', Vol. 1837 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 102–126.

Bowen, J. P. and Hinchey, M. G. (2005), Ten commandments revisited: a ten-year perspective on the industrial application of formal methods, *in* 'FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems', ACM, New York, USA, pp. 8–16.

Boyer, R. S. and Moore, J. S. (1988), *A Computational Logic Handbook*, Academic Press, Boston, MA, USA.

Burstall, R. (1972), Some techniques for proving correctness of programs which alter data structures, *in* B. Meltzer and D. Michie, eds, 'Machine Intelligence 7', Edinburgh University Press, pp. 23–50.

Cock, D., Klein, G. and Sewell, T. (2008), Secure microkernels, state monads and scalable refinement, *in* C. Munoz and O. Ait, eds, 'Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)', Lecture Notes in Computer Science, Springer-Verlag. To appear.

Common Criteria (2006), 'Common Criteria for information technology security evaluation (CC v3.1)', `http://www.commoncriteriaportal.org/`. Link visited July 2007.

Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A. and Vardi, M. Y. (2007), Proving that programs eventually do something good, *in* 'POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages', ACM, New York, NY, USA, pp. 265–276.

Cook, B., Podelski, A. and Rybalchenko, A. (2006), Termination proofs for systems code, *in* 'PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation', ACM, New York, NY, USA, pp. 415–426.

Dalinger, I., Hillebrand, M. A. and Paul, W. J. (2005), On the verification of memory management mechanisms, *in* D. Borrione and W. J. Paul, eds, 'Proc. 13th IFIP Conference on Correct Hardware Design and Verification Methods (CHARME 2005)', Vol. 3725 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 301–316.

Daum, M., Maus, S., Schirmer, N. and Seghir, M. N. (2005), Integration of a software model checker into Isabelle, *in* G. Sutcliffe and A. Voronkov, eds, '12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)', Vol. 3835 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 381–395.

Davis, M., ed. (1965), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Raven Press, NY.

Dawson, J. E. (2007), Isabelle theories for machine words, *in* 'Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)', Electronic Notes in Computer Science, Elsevier, Oxford, UK. To appear.

de Roever, W.-P. and Engelhardt, K. (1998), *Data Refinement: Model-Oriented Proof Methods and their Comparison*, number 47 *in* 'Cambridge Tracts in Theoretical Computer Science', Cambridge University Press.

Dennis, J. B. and Van Horn, E. C. (1966), 'Programming semantics for multiprogrammed computations', *Communications of the ACM* **9**, 143–155.

Derrin, P., Elphinstone, K., Klein, G., Cock, D. and Chakravarty, M. M. T. (2006), Running the manual: An approach to high-assurance microkernel development, *in* 'Proceedings of the ACM SIGPLAN Haskell Workshop', Portland, OR, USA.

DiBona, C., Ockman, S. and Stone, M. (1999), *Open Sources: Voices from the Open Source Revolution, Appendix A: The Tanenbaum-Torvalds Debate*, O'Reilly. `http://www.oreilly.com/catalog/opensources/book/appa.html`, Link visited May 2008.

Dörrenbächer, J. (2006), VAMOS microkernel, formal models and verification, *in* 'Talk given at the 2nd International Workshop on System Verification (SV 2006)', NICTA, Sydney, Australia. `http://www.cse.unsw.edu.au/~formalmethods/events/svws-06/VAMOS_Microkernel.pdf`. Link visited May 2008.

Duff, T. (1983), 'Duff's device', `http://www.lysator.liu.se/c/duffs-device.html`. Link visited May 2008.

Elkaduwe, D., Derrin, P. and Elphinstone, K. (2007), A memory allocation model for an embedded microkernel, *in* 'Proceedings of the 1st International Workshop on Microkernels for Embedded Systems', NICTA, Sydney, Australia, pp. 28–34.

Elkaduwe, D., Derrin, P. and Elphinstone, K. (2008), Kernel design for isolation and assurance of physical memory, *in* '1st Workshop on Isolation and Integration in Embedded Systems', ACM SIGOPS, Glasgow, UK, pp. 35–40.

Elkaduwe, D., Klein, G. and Elphinstone, K. (2007), Verified protection model of the seL4 microkernel, Technical report, NICTA. Available from `http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf`.

Elphinstone, K. (2004), Future directions in the evolution of the L4 microkernel, *in* G. Klein, ed., 'Proceedings of the 1st international Workshop on OS Verification 2004, Technical Report 0401005T-1', NICTA, Sydney, Australia.

Elphinstone, K., Klein, G., Derrin, P., Roscoe, T. and Heiser, G. (2007), Towards a practical, verified kernel, *in* 'Proceedings of the 11th Workshop on Hot Topics in Operating Systems', USENIX, San Diego, CA, USA.

Elphinstone, K., Klein, G. and Kolanski, R. (2006), Formalising a high-performance microkernel, *in* R. Leino, ed., 'Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)', Microsoft Research Technical Report MSR-TR-2006-117, Seattle, USA, pp. 1–7.

Feiertag, R. J. (1980), A technique for proving specifications are multilevel secure, Technical Report CSL-109, Computer Science Laboratory, SRI International, Menlo Park, CA, USA.

Feiertag, R. J. and Neumann, P. G. (1979), The foundations of a provably secure operating system (PSOS), *in* 'AFIPS Conference Proceedings, 1979 National Computer Conference', New York, NY, USA, pp. 329–334.

Feng, X. (2007), An Open Framework for Certified System Software, PhD thesis, Department of Computer Science, Yale University, New Haven, CT, USA.

Feng, X., Shao, Z., Dong, Y. and Guo, Y. (2008), Certifying low-level programs with hardware interrupts and preemptive threads, *in* 'Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)', ACM, New York, NY, USA. To appear.

Fetzer, J. H. (1988), 'Program verification: the very idea', *Communications of the ACM* **31**(9), 1048–1063.

Fischer, S. (2008), Formal verification of a big integer library, *in* 'DATE'08 Workshop on Dependable Software Systems'. Available from `http://busserver.cs.uni-sb.de/publikationen/Fi08DATE.pdf`.

Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G. and Clawson, S. (1996), Microkernels meet recursive virtual machines, *in* 'Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)', Seattle, WA, USA, pp. 137–151.

Gargano, M., Hillebrand, M., Leinenbach, D. and Paul, W. (2005), On the correctness of operating system kernels, *in* 'Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)', Vol. 3603 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 1–16.

Gödel, K. (1931), 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I', *Monatshefte für Mathematik und Physik* **38**, 173–198. English translation, 'On Formally Undecidable Propositions of Principia Mathematica and Related Systems, I', in Davis (1965), pp. 4-38.

Goerigk, W. and Hoffmann, U. (1998), Rigorous compiler implementation correctness: How to prove the real thing correct, *in* D. Hutter, W. Stephan, P. Traverso and M. Ullmann, eds, 'Applied Formal Methods—FM-Trends 98', Vol. 1641 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 122–136.

Goguen, J. A. and Meseguer, J. (1982), Security policies and security models, *in* 'Proceedings of the 1982 IEEE Symposium on Security and Privacy', IEEE Computer Society Press, New York, NY, USA, pp. 11–20.

Goguen, J. A. and Meseguer, J. (1984), Unwinding and inference control, *in* 'IEEE Symposium on Security and Privacy', IEEE Computer Society Press, New York, NY, USA, pp. 75–87.

Greenhills Software, Inc. (2008), 'Integrity real-time operating system', `http://www.ghs.com/products/rtos/integrity.html`.

Greve, D., Richards, R. and Wilding, M. (2004), A summary of intrinsic partitioning verification, *in* 'Fifth International Workshop on the ACL2 Prover and its Applications (ACL2-2004)'. Available from `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/greve-richards-wilding/acl2-paper.pdf`.

Greve, D., Wilding, M. and Vanfleet, W. M. (2003), A separation kernel formal security policy, *in* 'Fourth International Workshop on the ACL2 Prover and its Applications (ACL2-2003)'. Available from `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/greve-wilding-vanfleet/security-policy.pdf`.

Greve, D., Wilding, M. and Vanfleet, W. M. (2005), High assurance formal security policy modeling, *in* 'Proceedings of the 17th Systems and Software Technology Conference 2005 (SSTC'05)'.

Guttman, J. D., Herzog, A. L., Ramsdell, J. D. and Skorupka, C. W. (2005), 'Verifying information flow goals in security-enhanced Linux', *Journal of Computer Security* **13**(1), 115–134.

Haigh, J. T. and Young, W. D. (1987), 'Extending the noninterference version of MLS for SAT', *IEEE Transactions on Software Engineering* **13**(2), 141–150.

Hansen, P. B. (1970), 'The nucleus of a multiprogramming operating system', *Communications of the ACM* **13**(4), 238–250.

Hardin, D. S., Smith, E. W. and Young, W. D. (2006), A robust machine code proof framework for highly secure applications, *in* 'ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications', ACM, New York, NY, USA, pp. 11–20.

Hardy, N. (1985), 'KeyKOS architecture', *ACM SIGOPS Operating Systems Review* **19**(4), 8–25.

Harrison, J. (1996), HOL Light: a tutorial introduction, *in* M. Srivas and A. Camilleri, eds, 'Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)', Vol. 1166 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 265–269.

Harrison, J. (2006), Towards self-verification of HOL Light, *in* U. Furbach and N. Shankar, eds, 'Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR 2006)', Vol. 4130 of *Lecture Notes in Computer Science*, Springer-Verlag, Seattle, WA, pp. 177–191.

Härtig, H., Hohmuth, M. and Wolter, J. (1998), Taming Linux, *in* K. A. Hawick and H. A. James, eds, 'Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)', Springer-Verlag, Adelaide, Australia.

Heiser, G., Elphinstone, K., Kuz, I., Klein, G. and Petters, S. M. (2007), 'Towards trustworthy computing systems: Taking microkernels to the next level', *ACM Operating Systems Review* **41**(3).

Heitmeyer, C. L., Archer, M., Leonard, E. I. and McLean, J. (2006), Formal specification and verification of data separation in a separation kernel for an embedded system, *in* 'CCS '06: Proceedings of the 13th ACM conference on Computer and communications security', ACM, New York, NY, USA, pp. 346–355.

Heitmeyer, C. L., Archer, M., Leonard, E. and McLean, J. (2008), 'Applying formal methods to a certifiably secure software system', *IEEE Transactions on Software Engineering* **34**(1), 82–98.

Hennessy, J. and Patterson, D. (1996), *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, USA.

Hillebrand, M. A., in der Rieden, T. and Paul, W. J. (2005), Dealing with I/O devices in the context of pervasive system verification, *in* 'ICCD '05: Proceedings of the 2005 International Conference on Computer Design', IEEE Computer Society, Washington, DC, USA, pp. 309–316.

Hillebrand, M. A. and Paul, W. J. (2008), On the architecture of system verification environments, *in* 'Hardware and Software: Verification and Testing', Vol. 4899 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 153–168.

Hoare, C. A. R. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**(10), 576–580.

Hohmuth, M. and Härtig, H. (2001), Pragmatic nonblocking synchronization for real-time systems, *in* 'Proceedings of the General Track: 2002 USENIX Annual Technical Conference', USENIX Association, Berkeley, CA, USA, pp. 217–230.

Hohmuth, M. and Tews, H. (2003), The semantics of C++ data types: Towards verifying low-level system components, *in* D. Basin and B. Wolff, eds, 'TPHOLs 2003, Emerging Trends Track, Technical Report No. 187', Institut für Informatik, Universität Freiburg, Freiburg, Germany, pp. 127–144.

Hohmuth, M. and Tews, H. (2005), The VFiasco approach for a verified operating system, *in* 'Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems', Glasgow, UK. Available from `http://wwwtcs.inf.tu-dresden.de/~tews/Plos-2005/ecoop-plos-05-a4.ps`.

Hohmuth, M., Tews, H. and Stephens, S. G. (2002*a*), Applying source-code verification to a microkernel: the VFiasco project, *in* G. Muller and E. Jul, eds, 'Proceedings of the 10th ACM SIGOPS European Workshop', ACM, New York, NY, USA, pp. 165–169.

Hohmuth, M., Tews, H. and Stephens, S. G. (2002*b*), Applying source-code verification to a microkernel: the VFiasco project, Technical Report TUD-FI02-03-März 2002, Technische Universität Dresden, Dresden, Germany.

Huisman, M. and Jacobs, B. (2000), Java program verification via a Hoare logic with abrupt termination, *in* 'FASE '00: Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering', Vol. 1783 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 284–303.

in der Rieden, T. and Tsyban, A. (2008), CVM—a verified framework for microkernel programmers, *in* R. Huuck, G. Klein and B. Schlich, eds, 'Proceedings of the 3rd international Workshop on Systems Software Verification (SSV'08)', Vol. 217 of *Electronic Notes in Computer Science*, Elsevier, Sydney, Australia, pp. 151–168.

Jacob, J. (1989), On the derivation of secure components, *in* 'Proceedings of the IEEE Symposium on Security and Privacy', IEEE Computer Society, Washington, DC, USA, pp. 242–247.

Jacobs, B., Meijer, H. and Poll, E. (2001), 'VerifiCard: A European project for smart card verification'. Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI).

Jones, S. P. (2003), *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press.

Kaufmann, M., Manolios, P. and Moore, J. S. (2000), *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers.

Kemmerer, R. (1979), Formal verification of the UCLA security kernel: abstract model, mapping functions, theorem generation, and proofs., PhD thesis, University of California, Los Angeles, USA.

Kestrel Institute (1998), *Specware Language Manual*, Palo Alto, CA, USA.

Klein, G. and Nipkow, T. (2006), 'A machine-checked model for a Java-like language, virtual machine and compiler', *ACM Transactions on Programming Languages and Systems* **28**(4), 619–695.

Klein, G. and Tuch, H. (2004), Towards verified virtual memory in L4, *in* K. Slind, ed., 'TPHOLs Emerging Trends '04', Park City, UT, USA.

Kolanski, R. (2008), A logic for virtual memory, *in* R. Huuck, G. Klein and B. Schlich, eds, 'Proceedings of the 3rd International Workshop on Systems Software Verification (SSV'08)', Vol. 217 of *Electronic Notes in Computer Science*, Elsevier, Sydney, Australia, pp. 61–77.

Kolanski, R. and Klein, G. (2006), Formalising the L4 microkernel API, *in* B. Jay and J. Gudmundsson, eds, 'Computing: The Australasian Theory Symposium (CATS 06)', Vol. 51 of *Conferences in Research and Practice in Information Technology*, Hobart, Australia, pp. 53–68.

Leinenbach, D., Paul, W. and Petrova, E. (2005), Towards the formal verification of a C0 compiler: code generation and implementation correctness, *in* 'Third IEEE International Conference on Software Engineering and Formal Methods, SEFM 2005', pp. 2–11.

Leinenbach, D. and Petrova, E. (2008), Pervasive compiler verification—from verified programs to verified systems, *in* R. Huuck, G. Klein and B. Schlich, eds, 'Proceedings of the 3rd international Workshop on Systems Software Verification (SSV'08)', Vol. 217 of *Electronic Notes in Computer Science*, Elsevier, Sydney, Australia, pp. 23–40.

Leroy, X. (2006), Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant, *in* J. G. Morrisett and S. L. P. Jones, eds, '33rd symposium Principles of Programming Languages (POPL'06)', ACM, New York, NY, USA, pp. 42–54.

Leslie, B., Chubb, P., Fitzroy-Dale, N., Götz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y., Elphinstone, K. and Heiser, G. (2005), 'User-level device drivers: Achieved performance', *Journal of Computer Science and Technology* **20**(5), 654–664.

Leslie, B., van Schaik, C. and Heiser, G. (2005), Wombat: A portable user-mode Linux for embedded systems, *in* 'Proceedings of the 6th Linux.Conf.Au', Canberra, Australia. `http://lcs2005.linux.org.au`. Link visited May 2008.

Liedtke, J. (1995), On μ-kernel construction, *in* 'Proceedings of 15th ACM Symposium on Operating System Principles (SOSP)', Operating System Review 29(5), ACM, New York, NY, USA, pp. 237–250.

Lipton, R. J. and Snyder, L. (1977), 'A linear time algorithm for deciding subject security', *Journal of the ACM* **24**(3), 455–464.

MacKenzie, D. (2001), *Mechanizing Proof: Computing, Risk, and Trust*, The MIT Press.

Martin, W. B., White, P., Goldberg, A. and Taylor, F. S. (2000), Formal construction of the mathematically analyzed separation kernel, *in* 'ASE '00: Proceedings of the 15th IEEE International Conference on Automated software engineering', IEEE Computer Society, Washington, DC, USA, pp. 133–141.

McCauley, E. J. and Drongowski, P. J. (1979), KSOS—the design of a secure operating system, *in* 'AFIPS Conference Proceedings, 1979 National Computer Conference', Vol. 48, AFIPS Press, pp. 345–353.

Morgan, C. (1990), *Programming from Specifications*, 2nd edn, Prentice Hall.

National Institute of Standards and Technology (2002), 'Software errors cost U.S. economy $59.5 billion annually', `http://www.nist.gov/public_affairs/releases/n02-10.htm`. Visited April 2008.

Naur, P. and Randell, B., eds (1969), *Software Engineering. Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, Scientific Affairs Division, NATO, Brussels, Belgium.

Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N. and Robinson, L. (1980), A provably secure operating system: The system, its applications, and proofs. Second Edition, Technical Report CSL-116, Computer Science Laboratory, SRI International, Menlo Park, CA, USA.

Neumann, P. G. and Feiertag, R. J. (2003), PSOS revisited, *in* 'ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference', IEEE Computer Society, Washington, DC, USA, pp. 208–216.

Ni, Z., Yu, D. and Shao, Z. (2007), Using XCAP to certify realistic system code: Machine context management, *in* 'Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)', Vol. 4732 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 189–206.

NICTA (2006), 'L4/Wombat performance', Web page, `http://ertos.nicta.com.au/research/l4/performance.pml`. Visited May 2008.

NICTA (2008), 'L4/Iguana OS web site', `http://www.ertos.nicta.com.au/iguana/`. Visited May 2008.

Nipkow, T., Paulson, L. and Wenzel, M. (2002), *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *Lecture Notes in Computer Science*, Springer-Verlag.

Norrish, M. (1998), C formalised in HOL, PhD thesis, Computer Laboratory, University of Cambridge.

Norrish, M. and Slind, K. (1998–2006), *HOL-4 manuals*. Available at `http://hol.sourceforge.net/`.

Open Kernel Labs, Inc. (2007), 'OKL web site', `http://www.ok-labs.com`. Visited May 2008.

Owre, S., Rajan, S., Rushby, J., Shankar, N. and Srivas, M. (1996), PVS: Combining specification, proof checking, and model checking, *in* R. Alur and T. Henzinger, eds, 'Computer Aided Verification', Vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 411–414.

Parker, K. J. (2007), *The Escapement*, Orbit.

Perrine, T., Codd, J. and Hardy, B. (1984), An overview of the kernelized secure operating system (KSOS), *in* 'Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference', pp. 146–160.

Petrova, E. (2007), Verification of the C0 Compiler Implementation on the Source Code Level, PhD thesis, Saarland University, Computer Science Department, Saarbrücken, Germany.

Popek, G. J. and Farber, D. A. (1978), 'A model for verification of data security in operating systems', *Communications of the ACM* **21**(9), 737–749.

Popek, G. J., Kampe, M., Kline, C. S. and Walton, E. (1977), The UCLA data secure operating system, Technical report, UCLA.

Popek, G. J., Kampe, M., Kline, C. S. and Walton, E. (1979), UCLA data secure Unix, *in* 'AFIPS Conference Proceedings: 1979 National Computer Conference (1979 NCC)', AFIPS Press, pp. 355–364.

Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D. and Jones, M. (1989), Mach: A system software kernel, *in* 'Proceedings of the 34th Computer Society International Conference COMPCON 89'.

Reason, J. (1990), *Human Error*, Cambridge University Press.

Reynolds, J. C. (2002), Separation logic: A logic for shared mutable data structures, *in* 'Proc. 17th IEEE Symposium on Logic in Computer Science', pp. 55–74.

Ridge, T. (2004), A mechanically verified, efficient, sound and complete theorem prover for first order logic, *in* G. Klein, T. Nipkow and L. Paulson, eds, 'The Archive of Formal Proofs', `http://afp.sf.net/entries/Verified-Prover.shtml`. Formal proof development.

Robinson, L. and Levitt, K. N. (1977), 'Proof techniques for hierarchically structured programs', *Communications of the ACM* **20**(4), 271–283.

Rockwell Collins, Inc. (2003), *AAMP7r1 Reference Manual*.

Rushby, J. (1992), Noninterference, transitivity, and channel-control security policies, Technical Report CS-92-02, Computer Science Laboratory, SRI International, Menlo Park, CA, USA.

Saydjari, O., Beckman, J. and Leaman, J. (1987), Locking computers securely, *in* '10th National Computer Security Conference', pp. 129–141.

Schirmer, N. (2004), A verification environment for sequential imperative programs in Isabelle/HOL, *in* F. Baader and A. Voronkov, eds, '11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'04)', Vol. 3452 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 398–414.

Schirmer, N. (2006), Verification of Sequential Imperative Programs in Isabelle/HOL, PhD thesis, Institut für Informatik, Technische Universität München, Munich, Germany.

Schwichtenberg, H. (2004), Proof search in minimal logic, *in* B. Buchberger and J. A. Campbell, eds, 'Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings', Vol. 3249 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–25.

Scott, D. S. (1970), Outline of a mathematical theory of computation, *in* 'In 4th Annual Princeton Conference on Information Sciences and Systems', pp. 169–176.

Securityfocus (2008), 'Vulnerabilities web site', `http://www.securityfocus.com/vulnerabilities`. Visited May 2008.

Shapiro, J. S. (2006), Programming language challenges in systems codes: why systems programmers still use C, and what to do about it, *in* 'PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems', ACM, New York, NY, USA.

Shapiro, J. S. (2008), 'Coytos web site', `http://www.coyotos.org/`. Link visited May 2008.

Shapiro, J. S., Doerrie, M. S., Northup, E., Sridhar, S. and Miller, M. (2004), Towards a verified, general-purpose operating system kernel, *in* G. Klein, ed., 'Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification 2004. Technical Report 0401005T-1', NICTA, Sydney, Australia.

Shapiro, J. S., Smith, J. M. and Farber, D. J. (1999), EROS: a fast capability system, *in* 'SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles', ACM, New York, NY, USA, pp. 170–185.

Shapiro, J. S. and Weber, S. (2000), Verifying the EROS confinement mechanism, *in* 'Proceedings of the 2000 IEEE Symposium on Security and Privacy', IEEE Computer Society, Washington, DC, USA, pp. 166–176.

Slashdot (2006), 'Root exploit for NVIDIA closed-source Linux driver', `http://it.slashdot.org/article.pl?sid=06/10/16/2038253`. Visited May 2008.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. and Lepreau, J. (1999), The Flask security architecture: system support for diverse security policies, *in* 'SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium', USENIX Association, Berkeley, CA, USA, pp. 123–139.

Sridhar, S. and Shapiro, J. S. (2006), Type inference for unboxed types and first class mutability, *in* 'PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems', ACM, New York, NY, USA, p. 7.

Starostin, A. (2006), Formal verification of a C-library for strings, Master's thesis, Saarland University, Computer Science Department, Saarbrücken, Germany. Available at `http://www-wjp.cs.uni-sb.de/publikationen/St06.pdf`.

Starostin, A. and Tsyban, A. (2008), Correct microkernel primitives, *in* R. Huuck, G. Klein and B. Schlich, eds, 'Proceedings of the 3rd international Workshop on Systems Software Verification (SSV'08)', Vol. 217 of *Electronic Notes in Computer Science*, Elsevier, Sydney, Australia, pp. 169–185.

System Architecture Group (2003), 'The L4Ka::Pistachio microkernel', `http://l4ka.org/projects/pistachio/pistachio-whitepaper.pdf`. Link visited May 2008.

Tews, H. (2004), 'Verifying Duff's device: A simple compositional denotational semantics for goto and computed jumps', `http://wwwtcs.inf.tu-dresden.de/~tews/Goto/goto.ps`. Link visitied May 2008.

Tews, H. (2007), Formal methods in the Robin project: Specification and verification of the Nova micro-hypervisor, *in* H. Tews, ed., 'Proceedings of the C/C++ Verification Workshop 2007. Technical Report ICIS-R07015', Radboud University Nijmegen, Nijmegen, The Netherlands.

Tews, H., Weber, T., Poll, E., van Eekelen, M. and van Rossum, P. (2008), 'Formal Nova interface specification (Robin deliverable D.12)', `http://www.cs.ru.nl/~tews/Robin/specification-deliverable.pdf`. Revision 45. Link visited May 2008.

Tews, H., Weber, T. and Völp, M. (2008), A formal model of memory peculiarities for the verification of low-level operating-system code, *in* R. Huuck, G. Klein and B. Schlich, eds, 'Proceedings of the 3rd international Workshop on Systems Software Verification (SSV'08)', Vol. 217 of *Electronic Notes in Computer Science*, Elsevier, Sydney, Australia, pp. 79–96.

Tuch, H. (2008*a*), Formal Memory Models for Verifying C Systems Code, PhD thesis, School for Computer Science and Engineering, University of New South Wales, Sydney, Australia.

Tuch, H. (2008*b*), Structured types and separation logic, *in* R. Huuck, G. Klein and B. Schlich, eds, 'Proceedings of the 3rd International Workshop on Systems Software Verification (SSV'08)', Vol. 217 of *Electronic Notes in Computer Science*, Elsevier, Sydney, Australia, pp. 41–59.

Tuch, H. and Klein, G. (2004), Verifying the L4 virtual memory subsystem, *in* G. Klein, ed., 'Proceedings of the 1st International Workshop on OS Verification 2004, Technical Report 0401005T-1', NICTA, Sydney, Australia, pp. 73–97.

Tuch, H. and Klein, G. (2005), A unified memory model for pointers, *in* 'Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'05)', Montego Bay, Jamaica, pp. 474–488.

Tuch, H., Klein, G. and Heiser, G. (2005), OS verification—now!, *in* 'Proceedings of the 10th Workshop on Hot Topics in Operating Systems', USENIX, Santa Fe, NM, USA, pp. 7–12.

Tuch, H., Klein, G. and Norrish, M. (2007), Types, bytes, and separation logic, *in* M. Hofmann and M. Felleisen, eds, 'Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', Nice, France, pp. 97–108.

Verisoft Project (2006), 'BMBF-funding improves the security of computer systems', `http://www.verisoft.de/PressRelease20050826.html`. Visited May 2008.

Verisoft Project (2008), 'Verisoft repository', `http://www.verisoft.de/VerisoftRepository.html`. Visited May 2008.

Walker, B. J., Kemmerer, R. A. and Popek, G. J. (1980), 'Specification and verification of the UCLA Unix security kernel', *Communications of the ACM* **23**(2), 118–131.

White, P. and Allen Goldberg, a. F. S. T. (2002), 'Creating high confidence in a separation kernel', *Automated Software Engineering* **9**(3), 263–284.

Wiedijk, F. (2008), 'Formalising 100 theorems', `http://www.cs.ru.nl/~freek/100/`. Link visited May 2008.

Wiedijk, F., ed. (2006), *The Seventeen Provers of the World, Foreword by Dana S. Scott*, Vol. 3600 of *Lecture Notes in Computer Science*, Springer-Verlag.

Winskel, G. (1993), *The formal semantics of programming languages*, MIT Press, Cambridge, UK.